



*For **Anja** — who gives me all the time to build, write,
and keep learning.
With love and gratitude. ❤️*

Preface & History

Who this is for — developers who want to build more software than they can personally remember.

Read this if — you want the argument and the story behind it.

Skip this if — you have five minutes; take the [Rule Index](#), then [The Golden Path](#).

Takeaways

1. Building was never the bottleneck; maintenance is where fleets sink.
2. Automate verification before you automate creation — everything else follows.
3. Correctness lives in the repository's checks, not in the developer's memory.

Last updated: 2026-08-02.

An Unreasonable Idea

This book is built on an idea that sounds unreasonable: **a single developer can build — and, harder still, maintain — a whole fleet of focused, high-quality apps.** Not prototypes that rot after the demo. Real tools: an emoji picker with a hand-tuned TUI engine, a terminal graphics renderer with golden-file test suites, a documentation engine that prints its own books to PDF. Each one small, each one finished, each one still alive a year later.

The catch is that building was never the bottleneck. **Maintenance is where fleets sink.** Every app you ship adds a standing cost: dependencies drift, terminals change behavior, toolchains update, and your memory of *why* the code works fades. Multiply that by ten apps and the classic answer is: you can't. Pick one project, archive the rest.

Agentic development changes that math — but only if you set it up right. This book is the record of how, and the rules that made it hold.

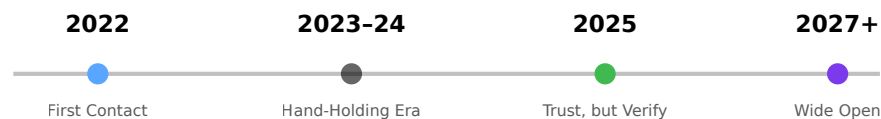
Two Principles

Everything else in this book unfolds from two sentences:

- **Engineering Principle:** Spec just enough, code for quality, verify and fail fast, iterate, succeed.
- **Agentic Principle:** Don't automate the creation before you automate the verification.

The second one is the load-bearing wall. An agent that can *write* code you cannot cheaply *check* is not leverage — it is unpaid review work arriving at machine speed. But once every app in the fleet carries its own verification harness — a `make preflight`, a canary script, a golden corpus — then any agent (and any future you) can safely touch any app, on any day, without re-loading a year of context first. **That is the whole trick. The fleet stays maintainable because correctness is checked by the repo, not remembered by the developer.**

The Journey



From a private-beta playground to a verified, local-first agent fleet.

2022 — First Contact

My journey with agentic development started in late 2022, playing in the OpenAI Playground with `code-davinci` in private beta. The completions were erratic, the context tiny — and I was hooked from day one, because even then the shape of the future was

visible: the machine could *write*; it just couldn't *know when it was wrong*.

The Hand-Holding Era

In the early days, I tuned my own tools from the start. Back then, agents needed constant supervision to get anything right. Every session was babysitting: watch the diff, catch the hallucinated API, re-explain the project. The lesson that stuck was not “agents are weak” — it was that **I was spending my time on checks a script could run**. I learned to build the guardrails, not just the prompts.

2025 — Trust, but Verify

The turning point: I stopped reviewing every step and started **verifying every result**. Automate the checks first, then let the agent run. A failed check costs nothing; an unverified merge costs a weekend. This inversion — verification before creation — is when one developer maintaining many apps stopped being a slogan and started being a Tuesday.

2027+ — Wide Open

Super curious where this leads. The models change every month, the harnesses change every week — and I want a front-row seat, with my repos ready for whatever shows up next.

Stance & Vision

The Problem: Harnesses are getting too big and too opaque. They eat developer time and hide what the agent is actually doing.

The Answer: Frontier models + a few generic helpers + many **manually triggered** skills and commands. The human triggers the steps; the human stays in control.

Models get better every month. Piling on complex harness code and endless context docs will age badly and do more harm than good. When a project goes spaghetti, a frontier model cleans it right back up — as long as the harness stays out of the way.

How to Read This Book

The chapters follow the same arc the fleet did:

1. The [Manifesto](#) states the rules; the [Glossary](#) pins the words down.
2. [The Golden Path](#) runs one real feature through every rule in this book — including the part where the agent is wrong and the harness is not yet watching. If you only believe demonstrations, start there.
3. [Canary-First Development](#) shows how we verify the *environment* before trusting it, and [Agentic Dev Patterns](#) collects the day-to-day workflow habits.
4. [TUI Restoration](#) and the [Golden-Change Playbook](#) are deep dives: what disciplined verification looks like when the problem gets genuinely hard.
5. [The Solo Fleet, Reviewed](#) closes the book by turning the lens around: where this CLI-first, local-first approach genuinely wins, where it would break inside a team or a product organization — and what those organizations should steal from it anyway. The [Colophon](#) then says how the book itself was made, and what the agents got wrong while making it.
6. The appendix, [House Rules](#), holds the language subsets for Go, Zig, Bash, and Make. It is reference material — grep it, don't read it.

Read it linearly for the story, or jump straight to the playbooks when something is on fire. Two pages are built for the reader in a hurry: the [Rule Index](#) is the whole book as one checklist, and every chapter opens with who it is for, what it assumes, and three takeaways.

Three signs recur throughout, and they carry the argument:

Sign	Meaning
 Agent takes the wheel	mechanical and verifiable — delegate it

Sign	Meaning
 Human holds the wheel	taste, architecture, irreversibility — keep it
 The check that makes it safe	the exact command that catches the agent being wrong

Every  in this book is followed by a . That pairing is not a formatting habit; it is the thesis.

The argument here is proved one shelf over: the [Emojig Chronology](#) is six weeks of the same practices with the commit counts, the deletions, and the millisecond measurements attached.

Rule Index

Who this is for — the reader with five minutes, and the agent that needs the whole book compressed into a checklist.

Read this if — you want every normative rule in this book on one page.

Skip this if — you want the reasoning; each rule links to the chapter that argues it.

Takeaways

1. Every rule in this book appears here exactly once, with exactly one home chapter.
2. Rules with a command in the last column are enforced by machine; the rest are enforced by attention.
3. If a rule is not here, it is not a rule — it is an opinion in a paragraph.

Last updated: 2026-08-09.

Principles

#	Rule	Home	Enforced by
P1	Never ask an agent for feature code that has no automated way to be checked.	Manifesto	judgement
P2	Write the harness, canary, or test suite <i>before</i> the feature.	Manifesto	judgement
P3	The repository is the only source of truth; if	Manifesto	judgement

#	Rule	Home	Enforced by
	it is not in a file, it does not exist.		
P4	Keep the harness lightweight and transparent; the human triggers steps and stays accountable.	Manifesto	judgement
P5	Prefer strict, self-contained tools and static binaries over resident daemon architectures.	Manifesto	judgement
P6	Every line you do not write is a line no agent can misunderstand — keep it minimal.	Manifesto	judgement

The Loop

#	Rule	Home	Enforced by
L1	When a check is green but the result is wrong, add the missing check <i>before</i> fixing the bug.	Golden Path	judgement
L2	A commit carries code, tests, and spec together, in conventional-commit form.	Golden Path	review
L3	Record which model helped in a <code>Co-Authored-By:</code> trailer, not in prose.	Golden Path	<code>git log</code>

#	Rule	Home	Enforced by
L4	Run the preflight gate before ending a turn or handing over.	Agentic Dev Patterns	make preflight
L5	Before a context compaction or session end, write state, warnings, and next steps into <code>issues/</code> .	Agentic Dev Patterns	judgement

Canaries

#	Rule	Home	Enforced by
C1	If you cannot verify a mechanism with a fake, write a canary before writing the feature.	Canary-First	judgement
C2	A canary isolates one external mechanism — no application logic, no abstractions.	Canary-First	review
C3	Run the canary against the real environment and inspect its output directly.	Canary-First	make canary
C4	Document what the canary found; the finding is the design input for the feature.	Canary-First	review
C5	Never delete a canary — keep it as the cheapest regression check you own.	Canary-First	make canary

Specs & Configuration

#	Rule	Home	Enforced by
S1	No layout constants, UI labels, key bindings, or colors inside application code.	Agentic Dev Patterns	review
S2	Declare them in <code>spec/</code> files and embed them at compile time (<code>//go:embed</code> , <code>@embedFile</code>).	Agentic Dev Patterns	review
S3	Generate every derivative format from the master spec; never hand-maintain a copy.	Agentic Dev Patterns	review
S4	One component, one language — never maintain the same logic twice in two languages.	Agentic Dev Patterns	review
S5	Prefer self-contained binaries that exit and clean up over background daemons.	Agentic Dev Patterns	review

Golden Files

#	Rule	Home	Enforced by
G1	Shrink the repro to the smallest failing input, and find a control that does <i>not</i> fail.	Golden-Change Playbook	judgement

#	Rule	Home	Enforced by
G2	Prove the root cause with numbers, not screenshots; state it as a formula.	Golden-Change Playbook	judgement
G3	Run the real production code in diagnostic scripts; never reimplement it.	Golden-Change Playbook	review
G4	Write down which goldens will change <i>before</i> editing code.	Golden-Change Playbook	judgement
G5	If the actual diff set differs from the prediction, revert and re-diagnose.	Golden-Change Playbook	test diff
G6	Fix the layer where the wrong value originates, and delete downstream band-aids.	Golden-Change Playbook	review
G7	Assert the underlying invariant in a unit test, not just the output string.	Golden-Change Playbook	test suite
G8	Regenerate goldens deliberately, revert unpredicted changes, and inspect the new values.	Golden-Change Playbook	git diff
G9	Never use the update flag to make a red suite green.	Golden-Change Playbook	review

#	Rule	Home	Enforced by
G10	Commit tests, code, docs, and issue references together.	Golden-Change Playbook	review

Terminal & Teardown

#	Rule	Home	Enforced by
T1	Every exit path — selection, cancel, <code>SIGINT</code> / <code>SIGTERM</code> , panic — runs the full teardown sequence, in order.	TUI Restoration	PTY harness
T2	Disable mouse tracking first, then drain in-flight input with a CPR (<code>\x1b[6n</code>) handshake.	TUI Restoration	PTY harness
T3	Restore termios before erasing anything.	TUI Restoration	PTY harness
T4	Erase only the rows the TUI occupied (<code>\x1b[2K</code>); never <code>\x1b[J</code> or <code>\x1b[2J</code> .	TUI Restoration	PTY harness
T5	Park the cursor at the recorded start row and emit nothing after it.	TUI Restoration	visual check
T6	Emit <code>?1049l</code> only if you emitted <code>?1049h</code> ; keep separate inline	TUI Restoration	visual check

#	Rule	Home	Enforced by
	and alt-screen restore sequences.		
T7	Keep the <code>?</code> in DEC private modes, and disable auto-wrap (<code>\x1b[?7l</code>) during inline renders.	TUI Restoration	review
T8	Parse mouse input with a ≥ 512 -byte buffer, a multi-event loop, and a carry buffer for split events.	TUI Restoration	PTY harness
T9	On full-width rows, skip trailing <code>\x1b[K</code> — pending wrap makes it erase the last drawn cell.	TUI Restoration	golden ANSI
T10	Verify teardown visually on a VTE terminal, on Foot, and under tmux.	TUI Restoration	manual protocol
T11	Parse CPR responses with a character-by-character state machine so user keystrokes are never eaten.	TUI Restoration	PTY harness
T12	Distribute grid remainder cells across the first <code>R</code> columns so widths sum exactly to <code>w</code> .	TUI Restoration	unit test
T13	Write sandboxed tool output under the	TUI Restoration	<code>make</code> <code>canary</code>

#	Rule	Home	Enforced by
	user's home, never to the system <code>/tmp</code> .		
T14	Set <code>PDEATHSIG</code> immediately after spawning a subprocess so no child outlives its parent.	TUI Restoration	review

Verification You Can Trust

#	Rule	Home	Enforced by
V1	Before marking version-gated behavior Resolved, confirm its regression test actually ran — a test that can silently skip is not a check.	The Canary That Skipped Itself	<code>AGENTS.md</code>
V2	Never infer an external tool's version threshold from one installed copy — probe the binary that will actually run.	The Canary That Skipped Itself	review
V3	Shared process state a test forces (a <code>sync.Once</code> , a package-level flag) must be reset on cleanup, or later tests silently inherit it.	The Canary That Skipped Itself	test suite

#	Rule	Home	Enforced by
V4	When a bug's only symptom is inside a GUI the agent cannot observe, ask a human to check one change at a time.	The Canary That Skipped Itself	manual protocol

Inheritance & First-Shot Code

#	Rule	Home	Enforced by
I1	Review generated code's success paths first — a fallback that reports success is more dangerous than a crash.	The Single-Prompt Miracle	review
I2	When a fix makes an operation fail closed, verify the operation is still reachable on a stock install.	The Single-Prompt Miracle	manual protocol
I3	Write the issue spec — Summary, Motivation, Proposed Solution, Alternatives Considered — before handing a fix to an unfamiliar agent or harness.	The Single-Prompt Miracle	review
I4	Audit the documentation a template brought with it in the same session, or file it as an issue.	The Single-Prompt Miracle	review

House Rules — Go

#	Rule	Home	Enforced by
GO1	Default to the standard library; get approval before adding a dependency.	Go	review
GO2	No ORMs, no logging frameworks, no DI containers; <code>cobra</code> is the one permitted CLI library.	Go	review
GO3	No package-level mutable state — pass state explicitly via arguments or a state struct.	Go	review
GO4	<code>init()</code> only for static, side-effect-free registration.	Go	review
GO5	Wrap errors with <code>fmt.Errorf("module: %w", err)</code> and let them bubble to <code>main()</code> .	Go	review
GO6	Never <code>panic()</code> outside unrecoverable boot-time configuration.	Go	review
GO7	Functions return values; only explicit printer helpers write to stdout.	Go	review
GO8	Table-driven tests with <code>t.Run</code> , standard testing only, in the same package.	Go	make test
GO9	Go sources stay <code>gofmt</code> - clean.	House Rules	<code>make</code> <code>preflight</code>

House Rules — Zig

#	Rule	Home	Enforced by
Z1	Never hide allocations; pass an <code>Allocator</code> explicitly.	Zig	review
Z2	Hot paths get a zero-allocation core (stack buffers, prefetched data).	Zig	bench
Z3	Build utility binaries with <code>-O ReleaseSmall</code> ; keep them under 1 MB and 5 MB RSS.	Zig	build check
Z4	Split helpers into dedicated files and run them via <code>std.testing.refAllDecls(@This())</code> .	Zig	zig t
Z5	Treat pinned 0.16.0 API shapes as dated notes — re-verify against the current release.	Zig	comp



House Rules — Bash

#	Rule	Home	Enforced by
B1	Every script starts with <code>#!/usr/bin/env bash</code> and <code>set -euo pipefail</code> .	Bash	review
B2	Use <code>if test ...; never [...]</code> or <code>[[...]]</code> .	Bash	review
B3	Put <code>then / else / do</code> on their own lines and avoid semicolons.	Bash	review
B4	Double-quote every expansion, and fail loudly	Bash	review

#	Rule	Home	Enforced by
	on missing arguments ("\${1:?usage}").		
B5	Declare <code>local</code> first, assign second, so command substitution exit codes survive.	Bash	review
B6	Use alignment continuation for long pipelines and conditions instead of fixed indents.	Bash	review
B7	Use <code>printf</code> over <code>echo</code> for variables, and route pass/fail messages through helper functions.	Bash	review
B8	Stay mawk-portable: no 3-argument <code>match()</code> , no <code>strtonum()</code> , no <code>gensub()</code> .	Bash	review

House Rules — Make

#	Rule	Home	Enforced by
M1	<code>help</code> is the first target; bare <code>make</code> prints usage.	Make	<code>make</code>
M2	Mark targets phony with the  sentinel dependency instead of per-target <code>.PHONY</code> lists.	Make	review
M3	Every user-facing target carries a <code># / ##</code> description and is scraped into <code>make help</code> .	Make	<code>make help</code>

#	Rule	Home	Enforced by
M4	Use := for immediate values, ?= for overridable ones, and align the assignments.	Make	review
M5	Express build order as target dependencies, not as a script of steps.	Make	review
M6	Always invoke the locally built binary (./\$(BINARY)), never one from \$PATH .	Make	review

Rules with no command in the last column are the ones an organization has to staff. That gap is not an oversight; it is the honest boundary of this approach, and it is the subject of [The Solo Fleet, Reviewed](#).

Manifesto of Agentic Dev

Who this is for — anyone deciding how much rope to give a coding agent in their own repository.

Read this if — you want the five commitments the rest of the book unpacks.

Skip this if — you want them as a checklist; that is the [Rule Index](#).

Takeaways

1. Verification comes before creation, always — the rest is detail.
2. Context that is not a file in the repository silently expires.
3. You can delegate execution; you cannot delegate accountability.

We believe in a future where software is co-created by humans and AI agents. That future has two endings. In one, codebases degenerate into unmaintainable spaghetti that no human fully understands and no agent can safely change. In the other, every repository carries its own rules, its own checks, and its own memory — so that any collaborator, carbon or silicon, can pick it up cold and improve it.

We are building for the second ending. These are the principles that keep us on that path:



Five principles, one goal: a fleet any collaborator can safely change.

1. Always Verify First

“Don’t automate the creation before you automate the verification.”

- Never ask or allow an agent to write feature code unless there is a concrete, automated way to verify its correctness.
- Write the validation harness, the canary, or the test suite *before* writing the feature.
- Let the agent fail fast, iterate, and succeed against strict automated constraints.

Why it matters: unverified agent output is not productivity — it is deferred review debt, compounding at machine speed.

 **Agent takes the wheel** — writing the harness itself. Test tables, fixture files, and golden corpora are mechanical work with an unambiguous definition of done.

 **The check that makes it safe** — `make preflight`, run before the turn ends. If the harness the agent wrote cannot fail, it is not a harness; see [The Golden Path](#) for what that looks like when it goes wrong.

2. Local-First, File-Based Workflows

- The codebase is the sole source of truth for both humans and agents.
- Specifications, ADRs, issues, context glossaries, and agent instructions live as plain files (`AGENTS.md` , `CLAUDE.md` , `CONTEXT.md` , `docs/` , `issues/`) inside the repository.
- Avoid proprietary cloud dashboards, hidden prompt injection layers, and complex agent databases. If it is not in the repository, it does not exist.

Why it matters: context that lives in files survives every session reset, every model upgrade, and every vendor change. Context that lives anywhere else silently expires.

3. Human in the Loop, Human in Control

- AI models get smarter and more capable every month. Opaque, heavy agent frameworks and babysitting harnesses age poorly.
- Keep the harness lightweight and transparent.
- The human acts as the architect, making strategic decisions and manually triggering skills or commands. The agent executes targets. The human stays in control.

Why it matters: you can delegate execution, but you cannot delegate accountability. The person who answers for the software must be able to see and steer what builds it.

 **Human holds the wheel** — choosing which steps run at all. Triggering a skill, accepting a plan, and deciding that a green build is nevertheless wrong are not automatable, because they are where responsibility actually lives.

4. Canary-First Verifications

- When a feature relies on an external CLI tool, hardware resource, OS-level socket behavior, or container layout, build a **canary** first.
- A canary is a minimal, standalone verification script that lives outside the main validation suite and validates the external mechanism in the real environment.
- Run it, observe, document the findings, and keep it in the repository.

Why it matters: assumptions about the environment are the most expensive kind of wrong. A canary converts a weekend of debugging into a thirty-second check.

5. Keep it Minimal and Radical

- Avoid generic, bloated design patterns. Write simple, clean, readable code.
- Prefer strict, self-contained tools (e.g. Go, Zig, static binaries) over massive, resident daemon architectures.
- Limit language usage to the [House Rules](#) subset to keep context windows clean and logic bulletproof.
- One component, one language. A second implementation of the same logic is not a backup — it is a divergence waiting to happen.

Why it matters: every line you don't write is a line no agent can misunderstand, no context window must carry, and no future session must re-learn.

The sharpest version of this principle is a deletion. In [Week 4 of the Emojig Chronology](#), the most consequential commit of the week removed **2,450 lines** — the duplicate Go clone `mojigo` that shadowed the Zig picker and made every search-scoring tweak a two-language edit. Nothing was lost except the divergence risk.



Agent takes the wheel — carrying out the deletion and re-pointing every build, script, and doc reference at the surviving implementation. Tedious, wide, entirely mechanical.



The check that makes it safe — the surviving implementation's own test suite plus a release build across every target. A deletion is safe exactly when the thing that remains is still provably correct.

Glossary

Who this is for — readers hitting a term in this book that carries more weight here than it does elsewhere.

Read this if — a word looked oddly specific.

Skip this if — you are reading linearly; come back when something snags.

Takeaways

1. Most terms here name a *file* or a *command*, not an abstraction — that is deliberate.
2. Where a term has a common industry meaning, the entry says how this book narrows it.
3. If a word is load-bearing in an argument, it is defined here; if it is not here, it is being used ordinarily.

The working vocabulary of a local-first, agent-assisted fleet.

The Working Vocabulary

Fleet

The set of small, finished applications one developer keeps alive at the same time. A fleet is defined by maintenance, not by authorship: code you wrote and no longer verify has left the fleet. *Avoid*: portfolio, suite.

Harness

The automated machinery that decides whether a change is acceptable — tests, golden corpora, canaries, formatters, and the target that runs them. The harness is what an agent is measured against, and the reason its output can be trusted without line-by-

line review. In this repository the harness is `make preflight` and everything it depends on. *Avoid*: CI, pipeline (those are where a harness runs, not what it is).

Preflight Gate

The single command that runs the whole harness and answers green or red (`make preflight`: canary, unit and golden tests, formatting). Run before ending a turn, writing a handover, or committing. *Avoid*: build step.

Canary

A real external mechanism, exercised for real, cheaply, before or alongside the thing that depends on it. That is the invariant; the packaging varies, and this library uses the word in three related senses:

1. **The canonical form** — a minimal, standalone script that verifies one external mechanism (`mdbook`, `chromium`, a socket, a sandbox) before any feature is built on it. No assertions framework, outside the main suite, never deleted. See [Canary-First Development](#).
2. **The gate** — `make canary`, the preflight layer that runs those scripts. Sense 1 is the artefact; sense 2 is the target that invokes it.
3. **A canary test** — a case *inside* the unit suite that shells out to the real binary rather than a fake, when the mechanism is too small to deserve its own script. It inherits the role but not the isolation, and with it a failure mode the standalone form does not have: a test can skip, and a skipped check reports the same green as a passing one. See [The Canary That Skipped Itself](#).

Sense 3 is a compromise, not a synonym. If a mechanism is worth probing at all and the probe is not trivially cheap, prefer sense 1. *Avoid*: test, probe (unqualified).

Reel

A scripted terminal session (a `.reel` file driven by the Wayreel recorder) that launches a TUI in a virtual display, injects keystrokes, and captures the result for visual or PTY verification. A reel is to a terminal UI what a golden file is to a renderer. *Avoid*: recording, macro.

Golden File

A verified, checked-in reference output — PNG, ANSI dump, or HTML — that a test compares against. Trustworthy only in proportion to the discipline behind its last regeneration. *Avoid*: snapshot, fixture.

Golden Change

A deliberate update to golden files, performed under the seven-step discipline in [The Golden-Change Playbook](#): predict the diff set first, then regenerate, then verify that exactly the predicted files changed. A golden change that was not predicted is a bug being ratified. *Avoid*: snapshot update.

Handover

The file an agent writes before a context compaction or the end of a session: current state, files touched, open warnings, next steps, blockers. It lives in `issues/` and is the repository's substitute for a memory the model does not have. See [Agentic Dev Patterns](#). *Avoid*: summary, notes.

Spec-Driven

The practice of keeping configuration — labels, layouts, key bindings, colors, fonts — in declarative files under `spec/`, embedded at compile time, with every derivative format generated rather than hand-maintained. Code reads the spec; the spec never mirrors the code. *Avoid*: config-driven, data-driven.

Stop-Words Gate

The build-time check that scans every book's Markdown for forbidden terms listed in `spec/config.yaml` and aborts the build with file and line number on a match. It is how the books stay accurate about which models and tools were actually used. *Avoid:* linter, filter.

Teardown Contract

The strict, ordered sequence an inline TUI must execute on every exit path — mouse off, drain input, restore termios, erase only its own rows, park the cursor. Not a cleanup routine but a contract, because partial compliance corrupts the user's terminal. See [TUI Restoration & Safe Teardown](#). *Avoid:* cleanup, shutdown.

Inline TUI

A terminal user interface that renders inside the normal scrollbar buffer instead of switching to the alternate screen — which is what makes the teardown contract necessary in the first place. *Avoid:* full-screen TUI.

House Rules

The per-language subset this project writes in (Go, Zig, Bash, Make), chosen so a cold reader can establish correctness locally. Argued in [How to Read the House Rules](#). *Avoid:* style guide, lint config.

Sentinel (⚙️)

The Unicode gear used as a dependency in Makefile targets to make them phony without maintaining a `.PHONY` list, and as the marker the self-documenting `help` target scrapes. *Avoid:* marker, flag.

local-first

Development patterns and tools that run entirely on the developer's workstation, with no dependency on cloud dashboards or online-only services. Its stronger form in this book: *if it is not in the repository, it does not exist*. *Avoid*: offline mode, self-hosted.

ADR (Architectural Decision Record)

A short Markdown file capturing one architectural decision, its context, and its rationale, numbered sequentially (`0003-spec-baseline-theme-cascade.md`). The artefact produced when a question is settled by judgement rather than by a check. *Avoid*: design doc, RFC.

Library Artefacts

Book

A multi-page Markdown document built with mdBook — this one, the [Emojig Chronology](#), and the Style Codex. *Avoid*: doc, manual.

Slide Deck

A Markdown presentation (slides separated by `---`) compiled into a single-page offline HTML app by `scripts/build_slides.go` . *Avoid*: slides, presentation.

Sync Assistant

A workflow — typically an agent session — that keeps book contents aligned with the code repositories they describe as those repositories change. *Avoid*: linter, builder.

Canary-First Development

Who this is for — anyone whose feature depends on a tool, a kernel behaviour, or a sandbox they did not write.

Read this if — you have ever lost a day to an assumption about the environment.

Skip this if — you already probe every external tool in isolation before you build on it.

Takeaways

1. Verify the mechanism alone, in the real environment, before building anything on it.
2. The canary's *finding* is the design input for the feature — write it down.
3. Never delete a canary; it is the cheapest regression check you will ever own.

Every project eventually bets on something it does not control: a CLI tool, a kernel behavior, a sandbox policy. Most debugging marathons are not caused by bugs in *your* code — they are caused by a wrong assumption about the environment, discovered three layers deep, at the worst possible time. Canary-first development moves that discovery to minute one.

A **canary** is a minimal, standalone verification script that validates one external mechanism before any feature code is built on top of it. It has no assertions framework and lives outside the main codebase validation suites. It runs against the real environment (filesystem, shell, external tool, network) and produces output you can inspect directly.

The name comes from the coal-mine canary: if the cheap verification check dies, you abort before investing in the tunnel.

When to Write a Canary

Write one whenever a feature depends on a mechanism that:

- Involves an external tool (`mdbook`, `chromium`, `grim`, `tesseract`, `ffmpeg`, `curl`, ...)
- Relies on OS-level behavior (PTY redirection, process substitution, socket lifecycle, file descriptor inheritance)
- Requires a specific environment (Wayland compositor, Xvfb display, GPU vs software render, snap packaging restrictions)
- Reads back its own output (tee capture, screenshot OCR, log scraping)

Rule of thumb: If you cannot verify it with a fake, write a canary before writing the feature.



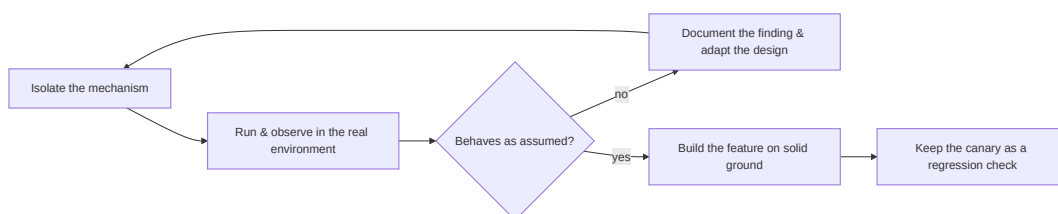
Agent takes the wheel — writing the canary. It is short, standalone, throwaway-shaped, and its success criterion is visible in the output. This is the single best first task to hand an agent on a new integration.



The check that makes it safe — `make canary`. It runs the real tool against the real filesystem and exits non-zero on the first surprise, so a wrong assumption costs thirty seconds instead of a weekend.

The Pattern

mechanism alone → observe output → assert manually → then build



1. **Isolate the mechanism:** Write the smallest possible script, reel, or program that exercises only the external piece. No

- application logic, no abstractions.
2. **Run it and observe:** Check the output file, log, or terminal directly. Does the mechanism do what you assumed?
 3. **Document what you found:** Note accuracy, failure modes, edge cases. This becomes the design input for the real feature.
 4. **Keep the canary:** Don't delete it. It is the cheapest future regression check and the fastest way for the next person to understand the mechanism.
-

Example — Snap Confinement (Our Own Canary)

During the creation of this book's PDF generation pipeline, we wrote a canary (`scripts/canary.sh`) that ran `chromium --headless --print-to-pdf` to verify that we could print our built book index HTML.

The Finding: The canary failed with: `Failed to write file /tmp/tmp.XXXX/test.pdf: No such file or directory`

Because Chromium is installed via Snap, it runs in a sandboxed mount namespace. It could not write to the OS-level `/tmp` directory. However, it had access to the user's home folder.

The Fix: We updated the script to write its temporary build outputs inside `/home/uwe/projects/books/.tmp/`. The canary passed immediately. This finding saved us hours of debugging inside the main application code!

Notice what *didn't* happen: nobody stepped through the PDF pipeline with a debugger, nobody blamed mdBook, nobody filed a bug against Chromium. A thirty-line script found the sandbox boundary before the feature existed — and it still guards that boundary today, every time `make canary` runs.



Human holds the wheel — deciding what the finding *means*. “Chromium cannot write `/tmp`” is an observation; “therefore every generated artefact in this project lives under

the repository root” is a design rule, and it has consequences an agent is not positioned to weigh.

The sequel proves the point better than the original. Years later the same project ran on a different distribution with a *flatpak* Chromium behind a shim — different vendor, different sandbox technology, same class of failure, caught by the same script. The full path from this finding to a shipped feature is walked step by step in [The Golden Path](#).

Not every canary stays this honest. [The Canary That Skipped Itself](#) walks through one that silently `t.Skip()` ed instead of failing when its tool wasn't on `PATH` — and let two wrong “fixes” ship as Resolved before anyone noticed the check had never actually run.

The Golden Path: One Feature, End to End

Who this is for — anyone who has read the rules in this book and wants to see them run once, on a real feature, in a real repository.

Read this if — you want to watch an agent build something, get it wrong, and get caught.

Skip this if — you only need the rules; they are in the [Rule Index](#).

Takeaways

1. The harness is written before the feature, and it is the harness — not the reviewer — that decides when the feature is done.
2. An agent's failure is usually not a wrong line of code; it is a *missing case* nobody wrote a check for.
3. A bug fix is not finished when the code is right. It is finished when the check that would have caught it exists.

Every other chapter in this book states a rule. This one runs a single feature through all of them, in order, in this repository:

The feature: *a book's fonts, size, leading, and accent color live in `spec/config.yaml` — nowhere else. Changing a book's look is a one-line spec edit.*

That is the feature that produced the stylesheet you are reading this page in.

On Evidence

What is real here, and what is reconstructed. Every file path, command, test name, commit hash, code excerpt, and tool output below is taken from this repository and can be checked with `git show <hash>` or by running the command yourself. The **prompts and session dialogue are reconstructed** — Git records what changed, not what was typed — and every reconstructed passage is labelled *illustrative*. Nothing in this chapter is a transcript pretending to be one.

Step 0 — Spec Before Code

The feature starts as data, not as code. `spec/config.yaml` gains a `books` section, one block per book:

```
books:
  main:
    font_family: "'Outfit', 'Inter', system-ui, sans-serif"
    font_size: "16.5px"
    primary_color: "#7c3aed" # Premium Royal Violet
    line_height: "1.6"
    margin: "1.2in"
```

Two things are already decided by writing this first. The generated stylesheet `src/custom.css` becomes an *output*, not a source file — it carries the header `/* Generated from spec/config.yaml - DO NOT EDIT MANUALLY */` and is not checked in. And the shape of the data is governed by `spec/config_schema.json`, so an editor flags a typo before a build ever runs.



Human holds the wheel — deciding *which* values belong in the spec. “Accent color, font stack, leading, margin” is a taste-and-architecture call: put too little in the spec and the theme drifts; put too much in and the spec becomes a

stylesheet in YAML costume. No check can make this decision.

Step 1 — Canary Before Feature

The feature is only interesting if the books actually build and print. Both depend on tools this repository does not control: `mdbook` and headless Chromium. So `scripts/canary.sh` runs first — it builds a three-file throwaway book and prints it to PDF:

```
make canary
```

The canary earned its keep immediately. Printing the throwaway book failed with

```
Failed to write file /tmp/tmp.XXXX/test.pdf: No such file or directory
```

Chromium was Snap-confined and could not see the OS `/tmp`. The fix was one line — write the temporary book into `./.tmp-canary` inside the project — and it is still in the script today. The full story is in [Canary-First Development](#); the repository's `issues/001` closed with the checkbox *“Fix snap sandbox issue for chromium by using a local `.tmp-canary` directory.”*

The interesting part is what happened *later*. The machine moved to Fedora with a flatpak Chromium behind a shim — a completely different packaging system, same class of failure.

`docs/Toolchain.md` records the resolution:

```
# allow headless captures to write into repo dirs (build/
etc.)
flatpak override --user --filesystem=home
org.chromium.Chromium
```

One thirty-line canary has now caught the same sandbox boundary twice, across two packaging technologies, years apart. That is the entire argument for canary-first compressed into one script.



Agent takes the wheel — writing and re-running the canary. It is mechanical, it is cheap, and the output is unambiguous: the script prints ✓ lines or exits non-zero.



The check that makes it safe — `make canary`. It fails loudly if `mdbook` is missing, if no Chromium is on `PATH`, if `mdbook build` produces no `index.html`, or if the printed PDF is zero bytes.

Step 2 — Harness Before Implementation

Now, and only now, is there a place to put a feature. `make preflight` is the gate, and it is composed, not monolithic:

```
preflight: 🌀 canary test styles-golden links rules render
website vendor-mermaid-verify canary-mermaid ## run all
gates
    @echo "==> Verifying code formatting..."
    ./scripts/check_format.sh
    @echo "✓ Preflight verification passed."
```

Every gate is a prerequisite or a script; the recipe itself decides nothing. That is deliberate. A make recipe joins continuation lines into a *single* shell line, so any real branching inside one collapses into the semicolon-heavy `if [ ... ]; then ... fi` layout that [Bash — House Rules](#) forbids. Logic that needs a conditional belongs in a script, where it can be written the way the rules say:

```
unformatted=$(gofmt -l "$TARGET")

if test -n "$unformatted"
then printf 'ERROR: Go files are not formatted. Run \''gofmt
-w %s'\''\n' "$TARGET" >&2
    printf '%s\n' "$unformatted" >&2
    exit 1
fi
```

`gofmt -l` prints the offending file names and exits `0` either way — the failure signal is the output, not the status. Converting one into the other is the script’s entire job.

Eight layers, each catching a different class of wrong:

Layer	Command	Catches
Environment	<code>make canary</code>	the toolchain moved, the sandbox changed
Behaviour	<code>go test scripts/build_slides.go scripts/build_slides_test.go</code>	the compiler renders the wrong thing
Presentation	<code>make styles-golden</code>	the theme changed without anyone deciding to change it
Links	<code>make links</code>	a dead cross-book link, or a <code>file://</code> path that only works on one machine
Normative rules	<code>make rules</code>	a rule-index ID that skips a number, a duplicated rule, an invented enforcement value
Rendered output	<code>make render</code>	prose that is correct in

Layer	Command	Catches
		Markdown and wrong in HTML
Placement	<code>make website</code>	a book whose <code>placement:</code> in the spec disagrees with the landing page
Hygiene	<code>scripts/check_format.sh</code>	code that will pollute the next diff

The hygiene layer is the youngest to come from a reader. Two chapters opened with a header box whose “Read this if / Skip this if” had folded into the paragraph above it — correct Markdown, wrong HTML, and invisible in every editor preview. No layer above could see it, because none of them read the rendered page. **A gate that never looks at the artifact the reader receives is not verifying the product.**

The behaviour layer is where the spec feature lives. Before the loader existed, the suite already held `TestGoldenRenders` — full slide decks rendered against checked-in golden HTML under `scripts/testdata/`. The feature adds `TestLoadSpecConfig_Books_DisableHeadersFooters`, which writes a miniature `config.yaml` into `t.TempDir()`, loads it, and asserts one field survived the round trip. No mocks, no framework, standard library `testing` — the house rule from [Go — House Rules](#).

Step 3 — The Agent Implements

The implementation is two functions in `scripts/build_slides.go`: `loadSpecConfig`, a line-by-line YAML scanner (the project’s dependency budget does not stretch to a YAML library for a forty-

line config), and `generateBooksCSS`, which writes one `custom.css` per book. `main()` wires them before anything else happens:

```
config, err := loadSpecConfig("spec/config.yaml")
if err == nil {
    g_config = config
    // Generate custom.css files for books styling
    if err := generateBooksCSS(config); err != nil { /* ...
exit 1 */ }
    // Validate stop words in source directories
    if err := validateSourceFiles(config); err != nil { /* ...
exit 1 */ }
}
```

That second call is the other half of the spec's authority:

`validateSourceFiles` walks `src`, `emojig/src`, and `styles/src`, and aborts the build if any line matches a pattern from the spec's `stop_words` array — with the file, the line number, and the offending text. The spec does not just style the books; it can refuse to let them build.



Agent takes the wheel — writing the scanner, the CSS emitter, and the table-driven tests. Bounded, mechanical, fully specified by the YAML above.



The check that makes it safe — `make test`, which runs `TestParseFrontmatter`, `TestFormatInline`, `TestRenderSlideToHTML_*`, `TestGoldenRenders`, `TestSlideTransitionCSS`, `TestParseYAMLValue`, and `TestLoadSpecConfig_Books_DisableHeadersFooters` against the real compiler.

Commit `1513f5d`, “*feat(spec): adopt spec-driven books and slides compilation with stop-words validation gate*”, shipped all of it. Preflight was green.

And the book was wrong.

Step 4 — The Agent Is Wrong

Here is the line that shipped, inside `loadSpecConfig`:

```
val := strings.Trim(strings.TrimSpace(parts[1]), `"'`)
```

It is a reasonable-looking line. It strips whitespace, then strips quotes from both ends. It handles `font_size: "16.5px"` perfectly. Now feed it the line the spec actually contained at that commit:

```
primary_color: "#7e57c2" # Sleek purple accent
```

`strings.Trim` removes quote characters from the *ends* of the string. This string does not end in a quote — it ends in a comment. Running the pre-fix expression over the pre-fix spec lines reproduces the damage exactly:

```
in :  "#7e57c2" # Sleek purple accent
old: #7e57c2" # Sleek purple accent

in :  "'Outfit', 'Inter', sans-serif"
old: Outfit', 'Inter', sans-serif
```

(Reproduced by running the pre-fix expression on the pre-fix spec lines — the values, not a memory of them. The formatting of this listing is mine.)

Two distinct failures from one line. The color became a string containing a quote, a hash, and four words of English, emitted straight into `--primary-color`. And the font stack lost its *leading* quote while keeping the rest, so the first family in the list was the unquoted garbage `Outfit',`.

Both are silently invalid CSS. Browsers do not raise; they discard the declaration and fall back. The book still built. The canary still passed. The golden tests still passed — because the golden corpus covered *slide rendering*, and nothing at all covered *spec parsing*.

 **This is the honest shape of an agent failure.** Not a hallucinated API, not a syntax error, not something a linter sees. A correct-looking one-liner that is right for the cases someone thought about and wrong for the case sitting in the config file three directories away. The harness did not catch it,

because the harness had no test for it. What caught it was a human looking at a page whose colors were quietly wrong.



Human holds the wheel — noticing that the rendered book looks off. No assertion in this repository was watching that value. Taste is still a detector, and it is the one you cannot delegate.

Step 5 — Iterate: Fix the Case, Then Fix the Blind Spot

Commit `79eb8fa`, “*feat(theme): enhance book layout styling and fix yaml parser quote/comment handling*”, does two things, and the order matters.

First, the parse is extracted into a named function with the comment rule ahead of the quote rule:

```
// parseYAMLValue parses a single line value from YAML,
stripping comments and matched quotes.
func parseYAMLValue(raw string) string {
    if idx := strings.Index(raw, " #"); idx != -1 {
        raw = raw[:idx]
    }
    raw = strings.TrimSpace(raw)
    if len(raw) >= 2 {
        if (raw[0] == '"' && raw[len(raw)-1] == '"') ||
        (raw[0] == '\'' && raw[len(raw)-1] == '\') {
            raw = raw[1 : len(raw)-1]
        }
    }
    return raw
}
```

Note `" #"` with the leading space, and the *matched-pair* quote test rather than a character set. Both are deliberate:

`value#no_space` is a value, not a comment, and `"'Outfit', 'Inter'"` must lose exactly one quote from each end.

Second — and this is the part that makes it a fix rather than a patch — the same commit adds `TestParseYAMLValue`, a table with the two spec lines that actually broke plus the edge cases the first attempt implied:

```
{`"'Outfit', 'Inter', sans-serif`, ` 'Outfit', 'Inter', sans-serif`},  
{`"#7e57c2" # Sleek purple accent`, `#7e57c2`},  
{`'single quotes'`, `single quotes`},  
{`no quotes`, `no quotes`},  
{`value#no_space`, `value#no_space`},  
{`value # comment with leading spaces`, `value`},
```

The blind spot is now covered. Restore the old one-liner today and `make test` fails within a second, naming the exact input and the exact expectation.



Agent takes the wheel — turning the observed failure into a table of cases and making the table pass. This is the loop agents are genuinely best at: a precise oracle, fast feedback, no judgement required.



The check that makes it safe — `TestParseYAMLValue` in `scripts/build_slides_test.go`. It is the permanent memory of a bug that nobody has to remember any more.

Illustrative prompt — reconstructed, not a transcript. The instruction that turns this from a patch into a fix is the second sentence:

The generated `src/custom.css` has `--primary-color: #7e57c2" # Sleek purple accent`. Find where the spec value is parsed and fix it. Then add a table-driven test that fails against the current code, including the real lines from `spec/config.yaml` and the `value#no_space` case. Run `make preflight`.

Step 6 — The Failure the Harness Cannot Catch

One release later the same feature broke again, at an altitude no unit test reaches. The Style Codex book had its own hand-written theme, `styles/css/fancy.css`, with a gold accent and a serif stack. The generated `custom.css` had the spec's values. Both loaded. The generated file won every shared selector, and the book rendered — per the record — as *“an incoherent pink/gold hybrid”*.

Every test passed. Every check was green. Nothing was broken in the sense a harness understands; two correct files simply disagreed about who was in charge.

The resolution was not code but a decision, written down as an architectural decision record (`docs/adr/0003-spec-baseline-theme-cascade.md`): the spec is the baseline, the theme refines it. Load the generated `custom.css` **before** the theme so the theme wins ties by cascade order, with no `!important` anywhere. Themes must not restate spec values — `fancy.css` derives its palette with `--codex-accent: var(--primary-color, ...)` and lightens it for dark themes via `color-mix()`. Rules the baseline already provided were *deleted* from the theme, not overridden.

The ADR closes with a sentence that is really a rule:

Theme rules that duplicate the generated baseline are treated as bugs.



Human holds the wheel — arbitrating between two systems that are each internally correct. This is architecture: the output is a rule and a written rationale, and its value is that the *next* agent inherits the decision instead of re-litigating it.

Step 7 — Preflight, Then Commit

The turn ends the same way every turn ends:

```
$ make preflight
==> Running books toolchain verification script (canary)...
✓ mdbook: ...
✓ chromium: ...
✓ mdbook build generated HTML
✓ chromium printed PDF successfully
==> Running Go unit and golden tests...
...
==> Verifying code formatting...
✓ Preflight verification passed.
```

(Shape of the real output; the version strings and PDF size depend on your machine. Run it and see yours.)

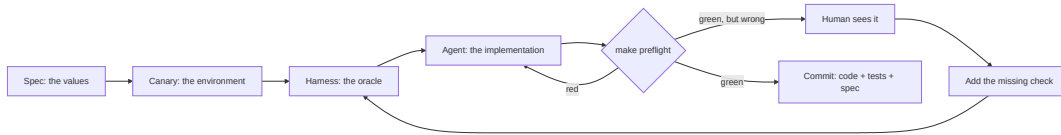
Then one commit carrying code, tests, and spec together, in conventional-commit form, with the model that helped recorded in a trailer:

```
feat(theme): enhance book layout styling and fix yaml parser
quote/comment handling

scripts/build_slides.go      | 77
+++++-----
scripts/build_slides_test.go | 23 ++++++++
spec/config.yaml             | 14 ++++----
```

Attribution is not decoration. At the time of writing, 16 of this repository's 37 commits carry a `Co-Authored-By:` trailer naming the model that worked on them; the earlier ones predate the convention. When a change looks strange a year from now, the trailer is the difference between archaeology and a `git log` query. Which models are in play at all is itself spec-governed — the registry lives in the `models:` array of `spec/config.yaml`, and the [Colophon](#) says what each did here.

The Loop, In One Picture



The branch that matters is the middle one. A harness that is only ever *red* or *green* describes a world where the tests are complete, and no test suite is. The third outcome — **green and still wrong** — is where the human belongs, and the only correct response to it is not “fix the bug” but “*add the check, then fix the bug.*” Every trip around that middle branch permanently shrinks the territory a human has to watch.

That is what “don’t automate the creation before you automate the verification” means on a Tuesday. Not that verification must be perfect before you start — it never is — but that every discovered gap gets closed in the harness before the turn ends, so the agent’s next run is safer than its last.

The Single-Prompt Miracle and What It Cost

Who this is for — anyone who has watched an agent produce a working application from one prompt and wondered what they just accepted.

Read this if — you build from templates, or you are about to ship first-shot generated code to a real user.

Skip this if — you never reuse a sibling project as a starting point, and every generated line you keep has already been read.

Takeaways

1. A single prompt produces an application only when a sibling project has already answered the structural questions — the speed comes from the template, not from the model.
2. First-shot code's dangerous defects are the ones that *report success*: a fallback, not a crash.
3. Inheritance is a loan. It arrives with the template's documentation, its assumptions, and its packaging bets, and those come due later.

`pdf-doctor` is a GTK4 application and a GNOME Files (Nautilus) context-menu extension that merges PDFs. It exists because of a tax deadline. It was built in one prompt, it worked, and it was wrong in three ways that no amount of running it would have shown.

This chapter is the audit, not the demo.

On Evidence

What is real here, and what is reconstructed. Every file path, commit hash, code excerpt, issue file, and file count below is taken from the `pdf-doctor` repository and can be checked with `git show <hash>` or by running the command shown. **The prompt in §1 is reconstructed** — Git records what changed, not what was typed — and is labelled *illustrative* where it appears. No dialogue in this chapter is presented as a transcript.

1. One Prompt, One Application

The problem was ordinary: a folder of receipts and statements that had to become a few categorised PDF bundles, some with specific page ranges, before a filing deadline. Uploading financial documents to a web converter was out. A multi-megabyte PDF suite was slower than the task. Hand-typing `pdfunite` invocations for every combination was the thing being avoided.

What was missing was ten seconds of workflow: select PDFs in Nautilus, right-click, **Merge...**, adjust ranges, save.

Illustrative prompt — reconstructed, not a transcript. The load-bearing clause is “using `../spriteview` as the template”; everything after it is a feature list:

Build `pdf-doctor` using `../spriteview` as the template: a GTK4 app plus a Nautilus extension that offers **Merge...** when two or more PDFs are selected, with drag-and-drop reordering, per-file page ranges (`1-3`, `5`, `8-12`), an engine choice, a headless `--merge` CLI, and an installer.

One commit came back:

```
$ git show --stat 9c6f28e | tail -3
tests/test_pdf_merge.py | 45 +++
website/index.html | 427
+++++
42 files changed, 3796 insertions(+)
```

It installed. Nautilus showed **Merge...** on a two-PDF selection. The bundles were built and the filing went out.

The interesting question is not “how did the model do that.” It is **what part of that did the model actually have to decide.**

2. The Speed Came From the Template

`spriteview` is a sibling in the fleet — a GTK4 tool for inspecting sprite sheets, with the same shape: a desktop window, a Nautilus entry point, a single-file packing step, an installer. Four structural questions were already answered there, and none of them were re-litigated:

Question	Where <code>spriteview</code> had already answered it
How does a GTK4 window get driven from a Nautilus extension without stalling the file manager?	<code>python3-nautilus</code> integration pattern
How does a multi-module Python app ship as one executable in <code>~/.local/bin</code> ?	<code>scripts/pack.py</code>
What are this codebase’s conventions?	<code>AGENTS.md</code> , <code>docs/</code> , the <code>harnez</code> block
What proves an install worked?	<code>make test</code> , <code>make install</code> , <code>make test-desktop</code>

The prompt did not ask for a desktop application architecture. It asked for a **domain-engine swap**: sprite rendering out, PDF page

slicing in. That is a bounded, mechanical task with a known-good scaffold — the single best shape of work to hand an agent, and the reason the output looked like a miracle instead of like a pile of broken C bindings.

State that plainly, because it is the transferable part: **single-prompt application generation is template instantiation**. Without `spriteview` in the sentence, the same prompt is a three-day project with a different failure mode every hour.

3. The Bill: Three Defects That Reported Success

Once the deadline passed, the generated code was read rather than run. Three issue files came out of that reading — `001`, `002`, and `003`, now in `issues/archive/`. All three share one property: **running the app could not have found them**, because in each case the wrong behaviour returns a success.

The first is the one worth printing. Here is `parse_page_range()` as it shipped in `9c6f28e`, ending with:

```
return pages if pages else list(range(1, max_pages + 1))
```

Read that line as a user of the feature. You typed `1-3` into a page-range field on a five-page bank statement. If your input parsed, you get pages one to three. If anything about your input failed to parse — a typo, a range past the end of the document, a stray character — `pages` is empty, and you get **all five pages**, silently, with a success dialog. The body of the function is built the same way: every parse error is `except ValueError: pass`, and out-of-range bounds are clamped into range rather than rejected.

`merge_pdfs()` compounded it one level up. If the selection engine was unavailable or failed, control fell through to the whole-file `pdfunite/gs` path — which cannot express a page selection at all, and therefore merged complete documents.

In a tax filing, the pages you deliberately excluded are the ones you most needed excluded. `issues/archive/001-preserve-requested-page-selection.md` states the principle in one sentence:

Returning a successful document with extra pages is worse than returning an error.

The fix in `0a31c9a` inverts the default: every malformed token raises, out-of-range pages raise instead of clamping, the input/range list lengths must match, engine arguments are built from *normalised page numbers* rather than from raw user text, and a custom selection that cannot be honoured stops with an error. Two more followed. `e674191` made publication safe — engines write a temporary file, the result is validated (non-empty, `%PDF-` magic) and only then published with `os.replace()`, and an output path that aliases an input is rejected up front, closing a path where the merge could destroy its own source. `29ac23f` made the installer create the directories it had assumed already existed.



Agent takes the wheel — turning each defect into a table of rejected inputs and making it pass.

`test_parse_page_range_rejects_invalid_input` subtests `"abc"`, `"0"`, `"6"`, `"1,2"`, `"1--3"`, `"2-6"`, and `"-"` against a five-page document, and

`test_custom_ranges_never_fall_back_to_whole_file_engines` collects every binary `subprocess.run` was handed and asserts each one ends in `qpdf` or `pdftk`.



The check that makes it safe — `make test` in `pdf-doctor`, which runs the syntax check, the unit tests, and the REUSE lint. The structural assertion is the important one: it checks *which binary was invoked*, not the returned message, so a whole-file engine can never again service a custom selection.

4. The Fix That Removed the Feature

Fail-closed was right, and it broke the feature for almost everyone.

Custom ranges were routed exclusively through `pdftk`. That was survivable while the code silently fell back to merging whole files

— the user got a wrong document, but they got a document. With the fallback correctly removed, every page-selection merge on a machine without `pdftk` now stopped with an error. `pdftk` is not in Fedora's repositories; it needs `pdftk-java` from a separate source. An advertised headline feature had become unreachable on a stock install — and it had *always* been unreachable there. Removing the fallback did not break it; it stopped hiding that it was broken.

`8b2a401` says so in its own message:

Since page selection correctly fails closed, the feature was unreachable without manually installing `pdftk`.

The repair adds `qpdf` — which ships in the core repositories of Fedora, Debian/Ubuntu and Arch, and has no 26-input handle limit — as the preferred selection engine, with `pdftk` as fallback and an error only when neither exists.

Then the same session nearly re-broke it. The `pdf_engine` setting names a *whole-file* engine (`auto` , `pdfunite` , `gs`). Filtering the selection engines by that setting means anyone who had chosen `pdfunite` in preferences silently lost page selection and was told no engine was installed. `docs/PDF_Engines.md` records both the trap and the two-line shape that avoids it:

```
if engine in ("qpdf", "pdftk"):
    selection_engines = (engine,)
else:
    selection_engines = ("qpdf", "pdftk")
```



Human holds the wheel — deciding that a correct error is still a broken feature. Nothing in the suite was red; every test of the fail-closed behaviour passed, on a machine where the maintainer had `pdftk`. What surfaced it was someone asking what happens on a fresh Fedora box. Packaging reality is a judgement call about users, and no assertion in the repository holds it.

This is the same failure as [The Canary That Skipped Itself](#), arriving from the other direction. There, one machine's `foot` version was

mistaken for every machine's; here, one machine's *package set* was. Rule V2 says to probe the binary that will actually run — its sibling is that **the machine which will actually run it is not yours.**

And it is the beat the “miracle” framing hides. The generated code was not merely incomplete — it had made a **packaging bet** nobody had noticed anyone making, and the bet only became visible once a correctness fix stopped papering over it.

5. Inheritance Is a Loan

The template gave the project its architecture. It also gave it sixteen documentation files describing another project.

That initial commit carried `docs/Packaging.md`, `docs/Interactive_Crop_And_Save.md`, `docs/In_Place_Navigation_And_Application_Uniqueness.md` and thirteen more — all seeded from `spriteview`, several describing features `pdf-doctor` does not have and never will. `docs/README.md` was still titled “*Sprite View - Documentation Index*” and linked every entry with an absolute `file://` URL, so the index of one repository pointed into another.

`issues/005-clean-inherited-spriteview-docs.md` is **still open**, and its motivation is the general form of the cost:

Stale docs are worse than missing docs: an agent or contributor reading `Packaging.md` (14 Sprite View references) cannot tell which statements still hold for PDF Doctor.

Note who the victim is. The documentation an agent reads to learn the project is the documentation most likely to have been inherited wholesale, because prose is what a template swap does not touch. Nine files under `docs/` still mention the wrong project — `grep -ril sprite docs/` counts them — and the index is the only one that has been rewritten.

The loan came due in a second currency too. `05df98a` exists because a stale installed copy of the extension made the running

behaviour disagree with the source — “a failure that cost three debugging rounds this session” — and its fix, `scripts/check-install.sh`, now compares the installed extension byte for byte against `dist/pdf_doctor.py` and reports when Nautilus started before the extension was installed.

 **Agent takes the wheel** — the byte comparison and the process-start-time check. Both are mechanical, and both encode a symptom that had already burned an afternoon.

 **The check that makes it safe** — `make check-install`, which fails when the installed artifacts differ from the packed build, and warns when no page selection engine is present at all.

6. The Guardrail Was the Issue File, Not the Harness

The three fixes in §3 were done as an experiment: a different CLI harness (**Goose**) driving a different vendor’s model, on a codebase neither had seen. The harness experience was rough — verbose, awkward interactive loops, no familiarity with the fleet’s conventions.

The fixes landed correctly anyway, and the reason is checkable. Each one was written as a specification file *before* execution, in the four-part shape `CONTRIBUTING.md` prescribes:

Section	What it pins down
Summary	the exact defect, in one paragraph
Motivation	why the current behaviour is unacceptable — here, silent page leaking
Proposed Solution	the architectural constraints: fail closed, <code>os.replace()</code> , create the directory

Section	What it pins down
Alternatives Considered	what is explicitly <i>forbidden</i> — “keeping the whole-file fallback was rejected”

The fourth section is the one that does the work. An agent that has been told what is forbidden cannot rediscover it as a clever simplification. Issue 001’s alternatives section names both the whole-file fallback and silent clamping, which are precisely the two shapes the original code had.

The experiment left one honest scar in the record. This book’s rule L3 (see the [Rule Index](#)) says to record the model in a `Co-Authored-By:` trailer. The Goose commits carry none:

```
$ git log --format='%h %s%n    %(trailers:key=Co-authored-by,valueonly)' --reverse | head -8
9c6f28e feat: initial commit for pdf-doctor GTK4 app and
Nautilus extension

0a31c9a fix(merge): preserve requested page selections

e674191 fix(merge): publish output safely

29ac23f fix(install): make repository setup reliable
```

Every commit after those four carries a trailer. The convention held wherever the usual harness ran and lapsed the moment it did not — which is the accurate version of “the tooling enforces it”: it does not. A person did, habitually, in one harness and not the other.

7. The Rules This Earned

Four, and they are all about the gap between *working* and *correct*.

Read the fallbacks first. In generated code, the paths that crash are the safe ones — someone will notice. Budget the review for every branch that returns success without doing what was asked. In `pdf-doctor` there were two, stacked, and one line of code held them both.

A correct error is still a broken feature. Making an operation fail closed changes who can use it. Check reachability on a stock install in the same session, or the fix ships as an outage.

Write the spec before you hand over the work, especially to an unfamiliar model or harness. The *Alternatives Considered* section is not documentation politeness; it is the fence.

Audit what the template brought. Code gets swapped, prose gets inherited, and inherited prose is what the next agent reads as truth. Two weeks after its first commit, `pdf-doctor` still carries nine documentation files about a sprite viewer.

The last one is the debt `pdf-doctor` has not finished paying. It is filed, open, and cited here rather than quietly omitted, because a chapter that audits an agent's output and hides its own outstanding balance is not an audit.

Full history is in `pdf-doctor: issues/archive/001 - 003` for the defects above, `issues/004` and `issues/005` for what is still open, and `docs/PDF_Engines.md` for the engine-selection reasoning.

Agentic Dev Patterns

Who this is for — developers already working with agents daily who want the habits that make the next session cheap.

Read this if — your agent keeps re-learning your project.

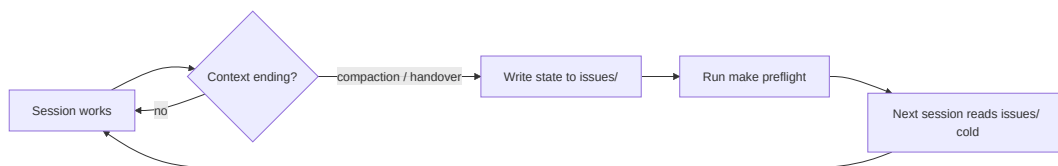
Skip this if — you have not yet built a harness; start with [Canary-First Development](#).

Takeaways

1. Treat the repository as the agent's memory: state, warnings, and next steps go into `issues/` before the context ends.
2. Configuration is data, not code — specs in `spec/` , embedded at compile time.
3. One component, one language; a duplicate implementation is a divergence with a schedule.

An AI coding agent is a brilliant colleague with a strange disability: at unpredictable moments, it forgets everything and starts over. To build software efficiently alongside such colleagues, our workflows must account for both model strengths and constraints (such as context window limits and formatting requirements) — and turn the repository itself into the memory the agent lacks.

The following practices are compiled from development cycles on `emojig` and `cati`.



1. Context Compaction & Self-Documentation

Frontier models lose fine-grained details when context windows are compacted or when starting new conversation sessions.

- **The Practice of Self-Documentation:** Before a context compaction event or at the end of a session, write down the current state, compiler warnings, design choices, and immediate next steps in a temporary or persistent issue tracker file (e.g., in `issues/`).
- **Preflight Cleanliness:** Always execute a standard check target (e.g., `make preflight`, formatter checks, and license linters) before ending a turn. This catches code formatting errors or compilation issues before they pollute the next agent's context.



Agent takes the wheel — writing its own handover.

Current state, files touched, open warnings, next step: the agent has this information and the human does not. Asking for it costs one instruction at the end of a session.



The check that makes it safe — `make preflight`

immediately before the handover file is written. A handover that describes a red repository as done is worse than no handover, and preflight is what makes the claim checkable.

2. Spec-Driven Architecture

Avoid embedding layout constants, UI labels, keyboard shortcuts, or color definitions directly inside application code.

- **Declarative Specs:** Define UI structure and bindings in declarative files (YAML or JSON) inside a `spec/` folder.
- **Compile-Time Embedding:** Load these specifications at compile time (using `//go:embed` in Go or `@embedFile` in Zig) to prevent runtime file reads.
- **Single Source of Truth:** Auto-generate derivative formats (such as JSON schemas or web simulator configurations) from

the master specifications. This prevents double-maintenance hazards where changes in the code must be manually copied to documentation or schemas.

This pattern was not designed on a whiteboard; it was extracted from a week of pain. [Week 2 of the Emojig Chronology](#) is where the picker's layouts, captions, key bindings, and colors moved out of the Zig loops and into `spec/` — after which **ten UI languages were added without a single line of Zig changing**. This book's own fonts and colors work the same way, and the story of the bug that shipped with them is [The Golden Path](#).



Agent takes the wheel — the extraction itself: hoisting constants into a spec file, generating the schema, and rewriting call sites. Wide, repetitive, and verifiable by construction.



The check that makes it safe — a schema-validated spec plus a unit test asserting the loaded struct matches it (here, `TestLoadSpecConfig_Books_DisableHeadersFooters` in `scripts/build_slides_test.go`), so a value that silently stops arriving fails the build instead of the page.

3. Complexity Reduction & Single-Language Focus

Supporting duplicate implementations of the same logic in different languages is a massive agentic maintenance burden.

- **Divergence Trap:** Maintaining a picker in both Go and Zig leads to silent differences in search scoring, layout boundaries, and terminal handling.
- **The Fix:** Consolidate on a single language for each component (e.g. rewriting and focusing 100% on Zig for the `emojig` TUI). Lowering repository lines of code simplifies the build pipeline and increases execution speed. The bill came due in [Week 4](#): 2,450 lines of Go clone deleted in one commit.
- **Zero-Daemon Philosophy:** Prefer statically linked, self-contained binaries that execute instantly and clean up after

themselves. Avoid background resident daemons that complicate error tracking and state synchronization.

Why These Patterns Scale to a Fleet

Each pattern above removes one reason a project needs *you specifically* to stay healthy:

- Self-documentation means any session — yours, an agent's, a future contributor's — can resume the work cold.
- Spec-driven architecture means style and structure changes are config edits, not code archaeology.
- Single-language focus means each repo asks for one mental model, not three.

Apply all three across ten small apps and the maintenance load stops multiplying: every repo greets its next visitor with the same files (`AGENTS.md` , `spec/` , `issues/` , a `make preflight`), the same checks, and no surprises. That uniformity — not heroics — is what lets one developer keep a fleet alive.

The Multi-Agent Fleet: Fresh Handoffs, Context Taxes, and the Observability Loop

How a Two-Week Crucible Turned Monolithic Agent Prompts into an Observable Multi-Agent Assembly Line

Who this is for — engineers and architects orchestrating AI coding agents across multi-repository fleets.

Read this if — your agent sessions are getting slow, bloated, prone to hallucination, or colliding on tickets.

Skip this if — you only run one-shot single-file prompt scripts.

Takeaways

1. **The Context Tax is Real:** Bloated monolithic sessions degrade agent attention (“lost-in-the-middle”); lean, ephemeral subagent handoffs (`/fresh-sprint`) slash token consumption by 70-80%.
2. **Telemetry Over Guesswork:** You cannot manage prompt drift without empirical metrics; mining Git history directly for document byte/token evolution (`harnetz dochistory`) provides the feedback loop needed for prompt dieting.
3. **Output Distillation Protects Context:** Massive compiler and test output drowns the model; automatic `PreToolUse` shell shimming (`harnetz distill`) preserves error lines while truncating noise.
4. **Green Unit Tests Can Lie:** Multi-agent pipelines require real-environment live validation gates; simulated test fixtures often hide runtime hook races and false workflow assumptions.

5. **Concurrency Demands Atomic Primitives:** When multiple agents operate in parallel, manual folk practices (`ls | grep`) fail immediately; atomic reservation (`harniez find issues next --reserve`) is essential.
-

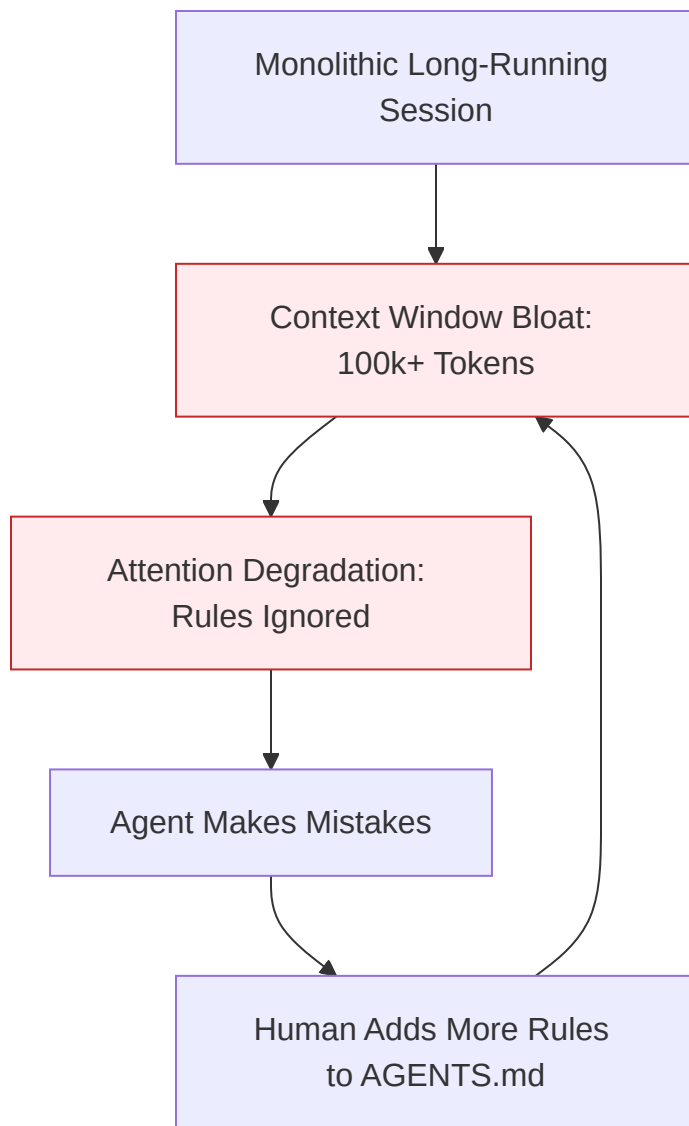
1. The Starting Point: The Monolithic Session Trap

In late August 2026, after months of operating an agentic software fleet across twenty repositories (`harniez` , `uman` , `voxi` , `ffext` , `cati` , `emoji`), we hit an invisible wall: **the monolithic session trap**.

In the early days of agentic coding, the common instinct was to keep a single chat session alive for as long as possible. The operator would feed task after task into the same conversation, accumulating hundreds of tool calls, compiler outputs, git diffs, and lint runs.

The failure mode wasn't a sudden crash. It was insidious:

- **The “Lost-in-the-Middle” Attention Cliff:** As the context window swelled past 80,000 tokens, the agent began silently ignoring explicit rules in `AGENTS.md` . It would invent ad-hoc commands, repeat fixed bugs, or replace fifty-line clean patches with fragile single-line edits.
- **The Compounding Token Tax:** Every single subsequent turn—even asking a one-word clarification—carried the full weight of the massive session history. A 10-turn simple bug fix cost 200,000+ tokens instead of 15,000.
- **Rule Accumulation & Prompt Bloat:** Every time an agent made a mistake, the team added another paragraph to `AGENTS.md` or `docs/lang/` . Over four months, instruction files doubled in size without anyone measuring what was actually being consumed.



2. The Breakthrough: The Lean Fresh-Handoff Sprint

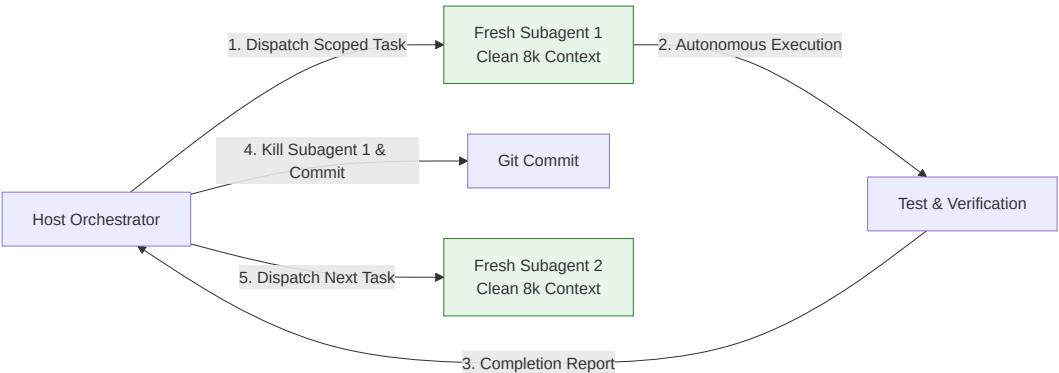
To break the cycle, we redesigned the agent workflow around **The Lean Fresh-Handoff Sprint (/fresh-sprint)**.

Instead of treating the AI assistant as a single eternal pair programmer, the host orchestrator acts as an **engineering supervisor**:

1. **Scoped Objective:** The host defines a razor-sharp, single-ticket objective with clear acceptance criteria and explicit non-goal fences.
2. **Ephemeral Subagent Dispatch:** A fresh subagent is launched in a pristine context window containing *only* the

baseline project rules and the target ticket.

- 3. **Autonomous Execution & Self-Verification:** The subagent implements the change, verifies it with repo-native test suites (`go test ./... , make check`), and reports back.
- 4. **Immediate Teardown:** The subagent is killed immediately upon task completion (`manage_subagents kill`). Zero zombie processes, zero background memory leaks.



Empirical Proof: The Three-Ticket Token Accounting Study

On August 28, 2026, we empirically measured a sequential run across three dependency-ordered tickets (Tickets 078 $\rightarrow$ 079 $\rightarrow$ 080: extracting the `rograph` layout module, adaptive graph shrinking, and panel boundaries):

Metric	Monolithic Session (Modeled)	Lean Fresh-Handoff Sprint (Actual)	Measured Delta
Total Tokens Consumed	$\geq$ 460,000 tokens	200,372 tokens	-56.4% (-260k tokens)
Tool Invocations	108 calls (accumulated)	108 calls (fresh contexts)	Identical work
Context Window at Task 3	$\sim$ 145,000 tokens	14,200 tokens	-90.2%

Metric	Monolithic Session (Modeled)	Lean Fresh-Handoff Sprint (Actual)	Measure Delta
Turnaround Time	~26 minutes	8.0 minutes	3.25x faster
Rework / Regressions	3-5 hallucinated turns	0 rework, 0 escalations	100% first-pass pass

3. Dynamic Context Pruning: From Injected Rules to Triggered Skills

Why had system prompts grown so large? On August 31, 2026, we ran a **Clean-Session Self-Audit** across Claude Code and Antigravity. By launching fresh sessions in an empty directory (`claude -p /` / `claude -c -p`), we measured every byte injected into the baseline system prompt.

The results were alarming:

- **Standing Policy Repetition:** The “Context Discipline” rule was injected four separate times with drifted wording across global, user, and project configurations.
- **Heavy Inactive Domains:** Massive domain guides (`Website.md` , `ConciseMode.md` , and the 1,500-token `Tool Feedback Protocol`) were being forced into every *single* prompt turn, even when the agent was just fixing a 2-line Go typo.

The Solution: Skills-Based On-Demand Ingestion

We stripped static prose from `AGENTS.md` and migrated domain guides into **Triggerable Agent Skills** (e.g. `~/claude/skills/website/SKILL.md`):

- **Always-on core:** Only core invariant boundaries (e.g., Make conventions, structured patch discipline, issue tracking).
 - **On-demand skills:** Heavy architectural guidelines are ingested *only* when the user or task explicitly invokes them (`/website` , `/concise-mode` , `/story`).
 - **Result:** More than **4,200 static tokens removed** from every baseline agent turn across the fleet.
-

4. Protecting Context: Tool Output Distillation (`harnez distill`)

Prompt instructions are only half the battle. The other half is **tool output bloat**.

When an agent runs `go test -v ./...` , `npm test` , or `git status` across a monorepo, the terminal tool output can dump 2,000 to 10,000 lines into the model's context. A single verbose test failure can consume 20% of the entire context window with irrelevant passed-test noise.

The PreToolUse Shell Interceptor

We introduced `harnez distill` and `harnez exec` :

1. **Hook Shimming:** Claude Code's `PreToolUse` hook intercepts shell execution before the command runs.
 2. **Transparent Auto-Piping:** Long-running or verbose commands are automatically routed through `harnez distill` .
 3. **Smart Error-Preserving Compression:** Distillation strips out verbose passing suites, collapses repetitive tracebacks, and preserves compiler errors and failed line numbers in a dense summary.
 4. **Byte-Cap Guardrails:** Guarantees that no single tool output exceeds safety bounds, preventing runaway token exhaustion.
-

5. Measuring the Beast: Git History Doc Telemetry (dochistory)

You cannot manage what you cannot measure. Having established that prompt bloat degrades LLM reasoning, we needed an empirical tool to answer: *How large are our managed documents, and how have they grown across Git history?*

Enter **harnes dochistory**: a zero-checkout telemetry engine capable of scanning dozens of documents across hundreds of Git commits in under **45 milliseconds**.

5.1 Streaming Git Architecture

Traditional tools (`git-of-theseus` , `hercules`) checkout working trees or maintain heavy in-memory DAGs. `dochistory` uses a streaming C-Git pipeline:

1. `git log --follow --raw --abbrev=40` streams commit metadata, file adds, deletions, and renames.
2. `git cat-file --batch` streams raw blob objects directly from Git packfiles without touching the working disk.
3. **Blob SHA-1 Caching**: Because Markdown files remain unchanged across many commits, metrics (`bytes` , `lines` , `words` , `tokens` , `headings`) are calculated **once per unique Blob SHA**. Cache hit rates exceed **80%**.

5.2 Visualizing Prompt Diet in the Terminal

`harnes dochistory` renders high-density 8-step Unicode sparklines () and dynamically filtered ANSI-colored stacked category proportion bars across monthly milestones:

— Multi-Doc Token Evolution & Stacked Category History

Documents: 15 analyzed | Commits: 66 | Unique Blobs: 108 |
Cache Hits: 9 (7.7%) | Latency: 41ms

Aggregate Sparkline:

[] (84 → 20,218
tokens | Peak: 20,639)

— Category Breakdown Across Milestones

Legend:  lang  practices  root (AGENTS.md)

MILESTONE

PROPORTION BAR

TOKENS DOCS

May 2026	[]
415 2	
Jun 2026	[]
3,340 7	
Jul 2026	[]
5,354 9	
Aug 2026	[]
19,817 15	
Present (01 Sep)	[]
20,218 15	

The output exposed the reality: in May, total documentation was 415 tokens. By late August, it had grown to 20,639 tokens—a **50x increase**. With `dochistory`, we could identify specific oversized practices (`AgenticLoop.md` at 5,124 tokens) and prune them systematically.

6. The Real-Environment Verification Gap: When Green Tests Lie

The greatest operational lesson of late August came from the **Tool Observability Saga (Issues 115-124)**.

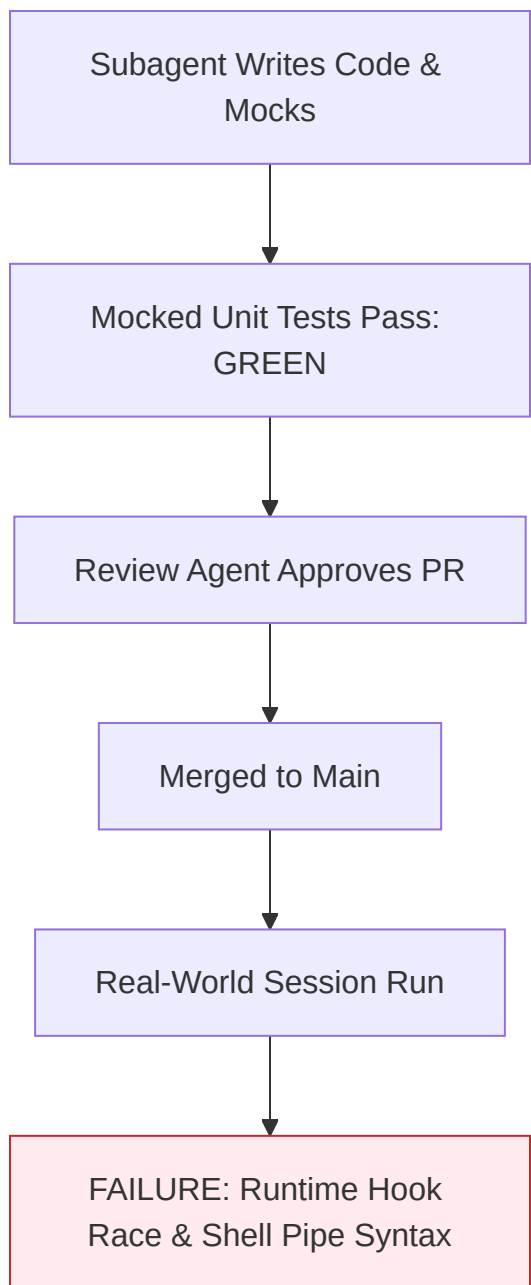
Over two days, eight sequential subagents implemented an automatic tool-call observability and telemetry pipeline. Every single subagent completed its task:

- 100% unit test coverage was achieved.
- `go test ./...` passed with zero errors.
- Code reviews approved every single diff block.

And yet, when the user started a real agent session, automatic telemetry capture was 100% dead.

What Went Wrong?

1. **Hook Racing in the Agent Runtime:** Claude Code executed multiple `PreToolUse` hooks concurrently. The last hook to finish overwrote earlier command rewrites without error.
2. **Metacharacter Shell Splicing:** Splicing raw commands into wrappers broke when commands contained `|`, `&&`, or nested quotes.
3. **Workflow Assumption Mismatch in Test Fixtures:** The unit test fixtures had mocked a Git workflow where agents operate on isolated ticket branches (`feature/116-telemetry`). In reality, the operator works directly on `main` . The code tried to parse branch ticket metadata that did not exist on `main` , silently aborting telemetry collection.



The Invariant: Mandatory Live-Environment Verification

This breakdown led to the creation of the **Real-Environment Verification Gate** in `docs/practices/AgenticLoop.md`:

- Unit tests prove that code satisfies *internal logic*.
- They **cannot prove** that runtime hooks, shell parsers, and external daemon processes cooperate under real human workloads.
- No infrastructure ticket is closed without a live-run verification in a restarted, real agent session with debug logging enabled (`DEBUG=true`).

7. Multi-Agent Concurrency: The Atomic Reservation Lesson

As multiple agents and parallel sessions began operating on the same repositories simultaneously, a subtle bug emerged: **ticket allocation races**.

Every agent had been computing the next issue number by running:

```
ls issues | grep -oE '^[0-9]+' | sort -n | tail -1
```

When two agents in parallel sessions (or an agent and a human) attempted to file tickets at the same time, both read number `194`, creating colliding files and broken Git history.

The Fix: `harnex find issues next --reserve`

We replaced the ad-hoc shell pipeline with a first-class CLI command:

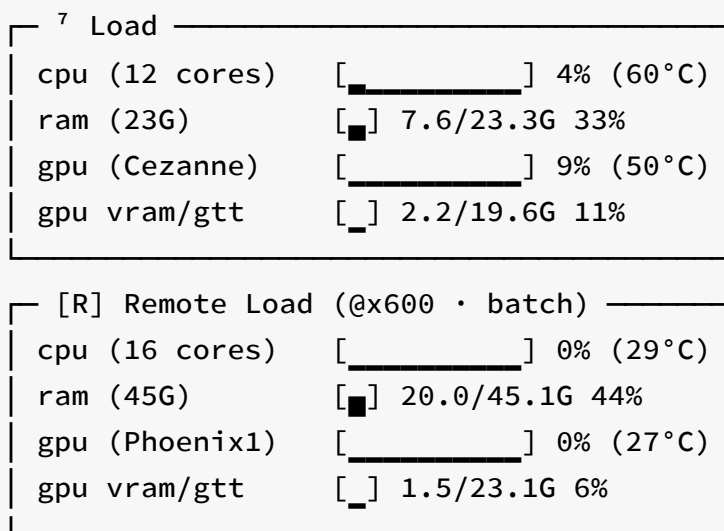
- `harnex find issues next`: Scans both active `issues/*.md` and archived `issues/archive/*.md` to calculate true `max + 1`.
 - `--reserve [title]`: Atomically claims the slot using `os.OpenFile(..., O_CREATE|O_EXCL)`, creating a placeholder draft ticket. A concurrent caller immediately receives `N+1`.
-

8. Live Observability: The Hardware Load Box Alignment

Agentic coding is CPU, GPU, and RAM intensive, especially when running local tokenizers, LLM sidecars, or compiling Rust and Go binaries.

In `harnex usage --watch` and `--compact`, we normalized the Hardware Load Box:

- **Zero-Subprocess Direct Kernel Reads:** Replaced heavy CLI forks (`nvidia-smi` , `rocm-smi` , `top`) with direct Linux `/proc` and `/sys` kernel telemetry.
- **RAM Geometry Discovery:** Replaced generic unpadded strings with hardware-aware labels (`ram (2x16G)` on workstation, `ram (45G)` on server node `@x600`).
- **Timeseries Sparklines for All Resources:** Unified CPU, GPU, RAM, and VRAM/GTT to display rolling 10-sample sparklines (`[██████]`), providing instant feedback on whether a subagent is active, thrashing, or idle:
- **Paced Startup Splash Screen:** Replaced the 3-second blank freeze on startup with an immediate alt-screen EWMA-estimated progress bar.



9. Polyglot Release Automation: From Go to WebExtensions

The final piece of the two-week evolution was **universal release ergonomics**.

`harnes release` originally targeted Go binaries with GoReleaser. In this session, we generalized it into a universal polyglot release engine:

1. **WebExtension & JSON Native:** Auto-detects `"version": "x.y.z"` in `manifest.json` and `package.json`, updating versions while preserving exact indentation.

2. **Monorepo Subproject Tagging:** Supports `tag_prefix:` `"linklit-v"`, allowing multi-extension repositories (`ffext`) to tag and release individual extensions independently (`linklit-v1.0.1` vs `lazypins-v1.1.2`).
3. **Build Target Fallback:** Executes `make pack` or `make dist` when no GoReleaser YAML is found.

Releasing any project in the fleet—whether a Go daemon, a Zig utility, or a Firefox extension—is now a single uniform command:

```
harnez release -d ~/git/ffext/linklit --bump patch
```

10. The Invariant Laws of the Multi-Agent Fleet

As we look across the last two weeks of intensive agentic pairing, six foundational rules have proven invariant:

1. **Kill Finished Contexts Immediately:** If a subagent completes its ticket, terminate it. Start the next task in a clean context to avoid the compounding context tax.
2. **Never Add a Rule Without a Metric:** Track documentation token weight over time. If a guideline doesn't prevent measurable errors, prune it.
3. **Offload Standing Prose to On-Demand Skills:** Keep baseline instructions minimal. Inject deep architectural rules only when specifically triggered.
4. **Distill Tool Outputs Before Ingestion:** Never let multi-megabyte test or build logs flood the model's context. Intercept and condense them.
5. **Real-Environment Gates Over Mocked Passes:** Unit tests verify code logic; only live execution with debug tracing verifies system integration.
6. **Automate Coordination Through Tools, Not Prose:** Do not ask agents to "be careful not to pick the same ticket number"; give them `harnez find issues next --reserve`.

Multi-Repo Agentic Workflows: Guardrails, Fast Classifiers, and the Sub-50ms Boundary

How a Cross-Project Lean Sprint Unified Neural Intent Classification with Deterministic CLI Performance

Who this is for — system architects, CLI designers, and AI engineers orchestrating autonomous multi-agent pipelines across multiple decoupled repositories.

Read this if — you want to bridge natural language intent into strict CLI interfaces without sacrificing millisecond-level responsiveness, or you manage multi-agent development workflows across repository boundaries.

Skip this if — you only write standalone single-repo scripts with no performance constraints.

Takeaways

1. **The Sub-50ms Boundary is Absolute:** Synchronous keystroke and shell tab-completion paths must remain 100% deterministic (<5 ms); model inference (~200-500 ms) belongs strictly at command invocation time.
2. **GBNF Grammar Constraints Eliminate Hallucination:** Constraining local model decoders with BNF grammars guarantees 100% schema validity, ensuring an agent or user argument maps strictly to a declared enum or explicitly abstains.
3. **The Zero-Coding Host Reviewer Pattern:** When a host orchestrator restricts itself to read-only diff inspection and delegates all file edits to low-cost worker agents, context bloat disappears and token velocity quadruples.

4. **Plan-First Read-Only Gating Catches Protocol Bugs**
Early: Forcing subagents to output a concrete file and test plan before granting write access catches missing flags, schema collisions, and test omissions before dirtying the working tree.
 5. **Clean Multi-Repo Separation Requires Stable CLI Contracts:** Separate tools (`harnez` , `neus` , `lmcoder`) should communicate over bounded JSON CLI contracts rather than internal library imports or shared memory.
-

1. The Architectural Dilemma: Natural Intent vs. CLI Rigor

As agentic developer tools evolve, a fundamental ergonomic tension emerges:

- **Strict Command-Line Interfaces** demand precise flags, canonical enums, and exact syntax (`harnez agent start --role developer`).
- **Autonomous Agents & Human Users** express intent in fuzzy, natural language (`harnez agent start --role "review the PR diff"` OR `harnez agent start explorer "audit the SQL queries"`).

When an agent guesses a near-miss argument (such as `--role explorer` instead of `--role advisor`), conventional CLIs fail abruptly with `unknown role "explorer"` . In a long multi-agent workflow, these tiny syntax collisions cause failed turns, retry loops, and human frustration.

The naive solution is tempting: *“Why not put an LLM in the shell’s auto-complete function or argument parser to dynamically classify every keystroke?”*

In this case study, we walk through the empirical research, the performance constraints that shattered this naive approach, and the robust multi-repo architecture built to solve it.

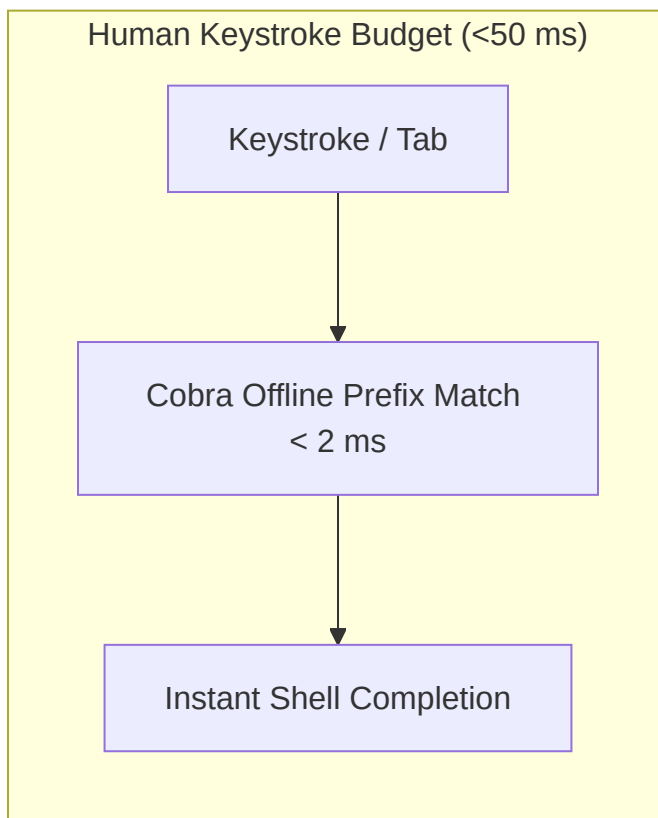
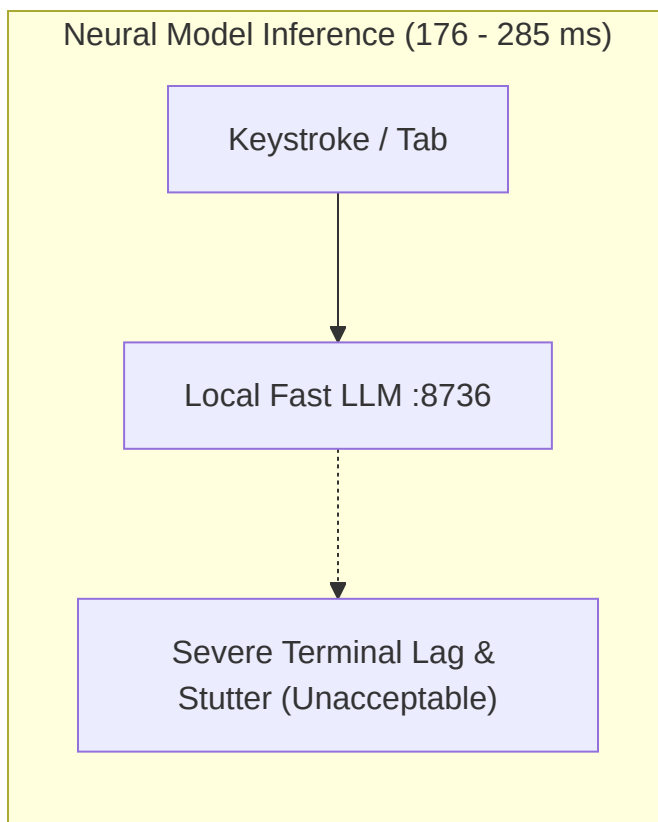
2. The Empirical Reality: The Sub-50ms Boundary

Before writing code, we dispatched parallel research agents across three repositories:

- `neus` : A Go-based neural search and AST-indexing engine.
- `lmcoder` : A local model lifecycle manager serving `llama.cpp` instances (`127.0.0.1:8736` fast chat, `:8734` primary chat, `:8746` embeddings).
- `harnez` : The multi-agent harness and developer CLI.

The research subagents benchmarked local small models (`Qwen2.5-0.5B` and `Qwen2.5-1.5B`) executing GBNF-constrained enum classification on local AMD APU hardware:

Model / Strategy	Latency (p50 / p95)	Cold Start	Schema Validity	Exact Match Rate
Qwen2.5-0.5B (GBNF Enum)	176 ms / 279 ms	703 ms	100%	34.8% (95.7% safe outcome)
Qwen2.5-1.5B (GBNF Enum)	285 ms / 410 ms	1.52 s	100%	56.5% (82.6% safe outcome)
Multi-field Extraction	1.45 s / 2.10 s	—	100%	70.0%



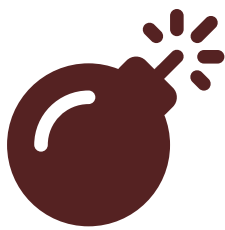
The Architectural Takeaway

Synchronous shell completion must never depend on neural inference.

Interactive tab-completion must remain strictly deterministic, instant (<5 ms), and offline. Neural classification belongs exclusively at **command invocation time**—resolving natural language after the user presses Enter.

3. The Multi-Repo Assembly Line: Step-by-Step Architecture

To implement this vision without coupling codebases, we established a strict contract boundary across `neus` and `harnez`.



Syntax error in text
mermaid version 11.16.1

1. `neus` classify: A Stable, Bounded Interface

Instead of importing internal Go packages across repo boundaries, `neus` exposes a clean CLI command:

```
neus classify --json --classes  
orchestrator,developer,reviewer,advisor "audit SQLite  
performance"
```

Output:

```
{  
  "class": "advisor",  
  "input": "audit SQLite performance",  
  "classes": ["orchestrator", "developer", "reviewer",  
"advisor"],  
  "model": "",  
  "latency_ms": 536,  
  "source": "model"  
}
```

If the input is an exact or unique prefix match (e.g. `"review"`), it short-circuits in <1 ms with `"source": "prefix"`. If the model is uncertain, it selects the private `unknown` grammar token and exits with code 2 (abstention).

2. harnez: The Dynamic Resolver Cascade

When `harniez agent start` is called:

1. **Exact match:** Checks canonical roles defined in `spec/agent.yaml`.
 2. **Static aliases:** Checks rich aliases (e.g. `coder` $\rightarrow$ `developer`, `explorer` $\rightarrow$ `advisor`).
 3. **Prompt Auto-Inference:** When `--role` is omitted, the prompt is sent to `neus classify` with a 1.5s bounded timeout. If classified as `reviewer` or `advisor`, Harnez prints an advisory notice and sets the role, falling back safely to `developer`.
 4. **Positional Ergonomics:** `harniez agent start explorer "audit queries"` automatically extracts the leading role token without requiring `--role`.
-

4. The Agentic Sprint in Action: Guardrails & Review Discipline

The implementation was delivered using the **Lean Fresh-Handoff Sprint** workflow. The Host Orchestrator operated under strict invariants:

Invariant 1: Zero Coding by the Orchestrator

The host never edited Go source files, YAML configs, or test fixtures. All changes were authored by dispatched `codex:luna:med` developer subagents. This kept the orchestrator's context window pristine and dedicated entirely to diff review.

Invariant 2: Plan-First Read-Only Gating

Every developer worker was first launched in read-only mode (`"read-only: plan ticket XXX..."`). The orchestrator reviewed the plan (schema fields, test cases, error paths) before issuing a `resume` command with write authority.

```
[Host]   harnez agent start --role developer --model
codex:luna:med "read-only: plan ticket 686..."
[Worker] Emits concrete plan: files, GBNF enum handling, test
mock strategy.
[Host]   Reviews plan -> "Plan approved; implement milestone
M1 now..."
[Worker] Implements, runs tests, commits, and reports back.
```

Invariant 3: Quota-1 Test Budget

Workers were bounded to a single full test suite run (`make test-q1`) before committing. When tests failed, the worker had to inspect the captured failure log, correct the root cause, and re-verify once rather than thrashing in an infinite loop.

5. Honest Post-Mortem: Real Bugs Caught in the Loop

No engineering system is built without friction. Here are the candid failures encountered and resolved during this session:



1. **The Goroutine Server Hang:** A simulated network timeout test spawned an HTTP test server that waited indefinitely, triggering Go's 10-minute test timeout. The worker diagnosed the stuck channel, bounded the handler to 1s, and re-verified cleanly.
2. **Abstention Assertion Inversion:** The developer added `unknown` as a private grammar token for model refusals, but wrote tests asserting that the public class list excluded

unknown . Reviewing the diff during milestone inspection caught the failure immediately.

3. **Legacy Test Pollution**: Adding prompt auto-inference caused existing unit tests to capture classifier logs on `stderr` . Injecting a mock classifier during test suite execution isolated the tests from ambient host model state.
-

6. The Result: Unified, Observable Ergonomics

By the end of the sprint, the entire subagent workflow was verified live:

```
# 1. Starting an agent with natural language prompt (inferred
role: advisor)
$ harnez agent start "audit the sqlite indexing queries in
neus"
harnes: inferred role "advisor" (aliases: explorer, planner)
from prompt
[session info: id=01a103... name=codex-01a103 role=advisor]

# 2. Positional role shorthand
$ harnez agent start review "check diff on HEAD~1"
[session info: id=01a104... name=codex-01a104 role=reviewer]

# 3. Fuzzy substring session resumption
$ harnez agent resume classify "run the CLI tests again"
harnes: matched session "neus-classify-dev" (id: 01a10277-
43c9...)
[resuming session neus-classify-dev...]

# 4. Contextual directory resumption
$ cd /home/uwe/projects/neus && harnez agent resume "continue
with M2"
harnes: resuming latest active session "neus-classify-dev"
(last active 2m ago)
```

7. Rules for the Multi-Repo Fleet

1. **The Performance Fence:** Any operation running on a keystroke or shell tab-completion hook must complete in under 5 ms. Never invoke neural models or external network sockets synchronously.
2. **The Grammar Guard:** When interfacing natural language with code execution, never parse raw unstructured LLM text. Constrain outputs with GBNF or JSON Schemas, and always provide an explicit `unknown` abstention path.
3. **The Orchestrator's Fence:** Keep host orchestrators out of the code editor. Review diffs, run verification targets, update tickets, and delegate implementation to low-cost, disposable subagents.

TUI Restoration & Safe Teardown

Who this is for — anyone shipping a terminal program that renders inline and has to give the terminal back intact.

Read this if — you are writing, reviewing, or debugging a TUI.

Skip this if — you write no terminal UIs; the specifics here do not generalize.

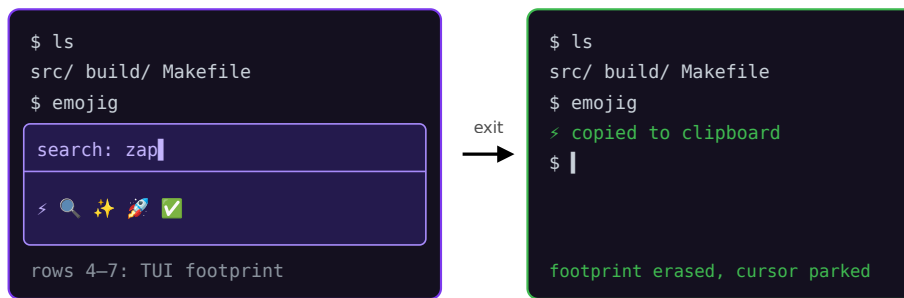
Takeaways

1. Teardown is a contract with a strict order, and every exit path — including panics and signals — must honour it.
2. Most terminal bugs are timing bugs: drain in-flight input before you restore, or the user's shell eats it.
3. Terminals disagree. Verify on VTE, on Foot, and under tmux, because no harness sees a cursor warp.

If you read one chapter of this book for craft rather than method, read this one.

Building inline TUIs (TUIs that render inline inside the normal terminal scrollbar rather than switching to the alternate screen buffer) requires surgical precision. A sloppy exit sequence leaves empty rows, corrupts terminal settings, or leaks escape sequences into the user's shell prompt. Users forgive a missing feature; they do not forgive a tool that wrecks their terminal on the way out.

The following conventions are derived from the development of `emojig`'s TUI engine — every rule below was paid for with a real bug on a real terminal.



The teardown goal: after exit, the scrollback looks like the TUI was never there.

1. The Inline Teardown Contract

On **every** exit path (normal selection, cancellations, signals like `SIGINT` or `SIGTERM`, and panic panics), the TUI must perform the following actions in strict chronological order:

1. **Disable Mouse Tracking:** Turn off mouse tracking sequences (e.g., `\x1b[?1003l\x1b[?1006l`).
2. **Drain In-Flight Input (CPR Handshake):** Before restoring cooked mode, write a Cursor Position Report query (`\x1b[6n`) and read the input stream until receiving the response (terminating with `R`). Because terminals process I/O in FIFO order, receiving this response guarantees that all mouse movement bytes queued *before* mouse tracking was disabled have been drained.
3. **Restore Termios:** Restore the raw terminal attributes back to the saved cooked mode state.
4. **Clean the Footprint:** Erase only the rows the TUI occupied. Loop through the rows and emit `\x1b[2K` (erase line). Do **never** use `\x1b[J` or `\x1b[2J` (erase display) as this destroys the user's terminal scrollback buffer.
5. **Park the Cursor:** Absolute-position the cursor (`\x1b[row;1H`) at the start row recorded at launch. Do not emit any character or escape code after this that could shift the cursor, preventing “stray blank lines” before the next shell prompt.

This contract was written in the first week of `emoji` — the very first commit established raw termios, signal safety, and the double-buffered surface before a single emoji was ever drawn. See [Week 1 of the Emoji Chronology](#) for the original code and the fourteen PTY-harness iterations it took to trust it.

 **Agent takes the wheel** — wiring the same teardown sequence into every exit path: normal return, cancel, `SIGINT`, `SIGTERM`, panic handler. It is exactly the kind of exhaustive, easily-forgotten coverage humans skip.

 **The check that makes it safe** — a PTY harness that launches the binary, sends each exit trigger, and asserts on the captured byte stream: mouse tracking off, CPR consumed, no `\x1b[J`, cursor parked at the start row.

2. Cross-Terminal Alt-Screen Pitfall (VTE vs. Foot/Tmux)

The private mode sequence `?1049h/l` switches to/from the alternate screen and saves/restores the cursor.

- **The Pitfall:** If your TUI runs inline (on the normal screen) and you emit `?1049l` during teardown just to be safe, GNOME Terminal, Tilix, Ptyxis, and other VTE-based terminals will still execute the **cursor-restore half** of the command, warping the cursor to a stale saved position and corrupting the next shell prompt line. Foot and tmux will silently ignore this.
- **Convention:** Only emit `?1049l` if you explicitly emitted `?1049h` to enter the alternate screen. Maintain separate `RESTORE` (inline) and `RESTORE_ALT` (alt-screen) sequences.

3. DEC Private Modes and ? Wrap Safety

Every terminal mode used for TUI rendering (`?25` cursor, `?1003 / ?1006` mouse, `?7` auto-wrap) is a DEC private mode.

- **The ? is load-bearing:** Omitting the `?` (e.g. sending `\x1b[7l` instead of `\x1b[?7l` to disable auto-wrap) results in

an unrecognized command or fires a completely different ANSI mode.

- **Wrap Disabling:** Always disable auto-wrap (`\x1b[?7l`) during inline renders. If an emoji or graphic wraps at the bottom row of the screen, the terminal will scroll, invalidating the recorded start coordinates and breaking the teardown row calculations.
-

4. Mitigating SGR Mouse Parser Leaks

When mouse tracking is active, the terminal emits SGR sequences (`\x1b[<Cb;Cx;CyM`) at high frequencies (up to 125 Hz). If the user scrolls or drags a scrollbar quickly:

1. **The Trap:** The read buffer might split an event (e.g., buffer ends at `\x1b[<32;32;`). The trailing bytes (`12M`) arrive in the next read *without* a leading escape byte, resulting in the TUI treating them as text input and typing them into the search box.
2. **The Fix:**
 - Use a larger read buffer (at least 512 bytes) to hold multiple full events.
 - Use a multi-event parser loop that processes all consecutive events in a single read.
 - Keep a carry buffer for incomplete events. If a read ends inside an SGR sequence, copy the fragment to the carry buffer and prepend it to the next read.
 - In panic handlers, perform a non-blocking drain of the input stream to clear the terminal's queue.

See [Emojig Week 4](#) for the dead end that paid for this rule — the week `12M` started appearing in the search box.

5. Exact-Width Pending-Wrap Erase Bug (`\x1b[K`)

- **Symptom:** The very last character of a row (such as a border cap `┌`) is drawn but is invisible on exact-width windows.

- **Root Cause:** Writing a character to the final column N leaves the cursor in a pending-wrap state. Running a standard `\x1b[K` (erase to end of line) from this state clears the character that was just drawn.
- **Fix:** For rows that span the entire terminal width, avoid trailing erase commands (`\x1b[K`). Emit only character reset codes (`\x1b[0m`).

See [Emojig Week 5](#) for how this one was found — a border cap that was drawn, present in the output bytes, and still invisible.

6. Visual Verification Protocol

Automated PTY harnesses cannot detect visual coordinate drift or VTE-specific cursor warps. Always visually verify teardowns on:

1. **A VTE Terminal** (e.g. GNOME Terminal) to verify alternate screen boundaries.
2. **Foot** to inspect standard-error logs for unimplemented sequences or bad modes.
3. **Tmux** to verify compatibility with terminal multiplexers.



Human holds the wheel — the eyes. A cursor warped by two rows, a flicker on resize, a prompt that lands one line too low: no assertion in any harness sees these, and they are precisely the defects users notice first.

One structural defence against escape-sequence rot is to stop scattering the literals. In [Week 5 of the Emojig Chronology](#), 100+ raw ANSI escapes were centralized into a `term.zig` utility module, moving 145 lines out of the rendering loops — the same week search latency fell from ~4.2 ms to ~0.36 ms per keystroke. Escape sequences you can only fix in one place are escape sequences an agent can fix safely.

7. CPR Handshake Race Condition & State-Machine Parser

When querying the cursor position during raw-mode startup or resizing using `\x1b[6n`, the response `\x1b[R0W;COLR` is sent by the terminal to standard input.

- **The Race Condition:** If the user is typing rapidly while the TUI launches or resizes, the user's keystrokes and the terminal's Cursor Position Report (CPR) bytes will interleave in the OS input buffer (e.g., `a\x1b[24;10Rb`).
 - **The Trap:** Relying on simple string comparison or regex matching to parse the CPR will fail or eat user keystrokes.
 - **The State-Machine Parser:** Process stdin character-by-character using a sequential state machine. Parse escape sequences into a dedicated struct and route any non-escape character directly to the TUI's command buffer or search query. This guarantees that no user input is lost and the CPR is parsed accurately.
-

8. Grid Layout Remainder Math Proof

When rendering a TUI grid (like an emoji picker or file browser) across a fixed terminal width `w`, columns are allocated a width `c`.

- **The Gutter/Overflow Problem:** If $w \bmod c \neq 0$, the remaining space $R = w \bmod c$ causes an empty column gutter on the right or layout overflow on the margins.
- **Remainder Distribution Invariant:** Distribute the `R` leftover cells across the columns as padding, widening the first `R` columns by one cell:

```
width(i) = C + 1    for the first R columns
width(i) = C        for the remaining columns
```

```
width(1) + width(2) + ... + width(N) = W
```

The widths sum back to exactly `w`, so the grid bounds snap to the terminal width without empty gutters or right-margin

9. Subprocess Lifecycle Management in Sandboxes

Running terminal tools or helpers inside sandboxes (like Snap confinement, Flatpak, or container mounts) restricts sub-process execution.

- **Mount Namespaces:** Sandboxed applications (e.g., Snap-confined Chromium) run in isolated mount namespaces and cannot write to the system `/tmp` directory. Always output files (such as temporary PDFs or assets) to directories within the user's home folder (e.g., `~/.cache` or `/home/uwe/projects/...`).
- **Orphaned Subprocesses:** If the parent TUI process crashes or receives a terminal signal, any spawned child process (like video pipelines, rasters, or encoders) will keep running, polluting the system resources.
- **PDEATHSIG Control:** On Linux, immediately after spawning a subprocess, set the process death signal attribute using raw system calls:

```
// In Go
syscall.Syscall(syscall.SYS_PRCTL,
syscall.PR_SET_PDEATHSIG, uintptr(syscall.SIGKILL), 0)
```

This instructs the OS kernel to automatically send `SIGKILL` to the child process the instant the parent thread dies, preventing orphaned processes.

The Golden-Change Playbook

Who this is for — anyone maintaining a golden/snapshot corpus, especially one an agent can regenerate.

Read this if — you are about to run a test command with `-update`.

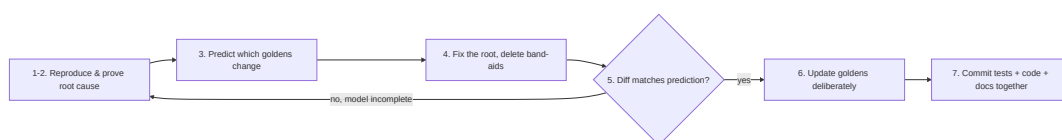
Skip this if — you have no golden files; the discipline still transfers, the commands do not.

Takeaways

1. Predict which goldens will change *before* you edit code; a mismatch means your model of the bug is wrong.
2. Prove the root cause with numbers and a formula, never with a screenshot.
3. `-update` is a ratification ceremony, not a way to make a suite green.

Visual golden files (like PNGs or ANSI snapshots) serve as the ground truth for what renderers produce. However, careless golden updates can easily ratify and hide bugs. A golden test suite is only as trustworthy as the discipline behind its last `-update` — one lazy regeneration and your safety net is quietly certifying the crash it was built to catch.

The following playbook, established during the development of `cati`'s terminal graphics engines, enforces the discipline required to fix visual regressions safely.



The Golden Rule: Prove the bug numerically before you touch a line of code, and change a golden image only once you can name the specific incorrect pixel/character and explain why it was wrong.

The Diagnostics & Update Loop

1. Reproduce & Isolate → Find the smallest input; find a control that does NOT fail.
2. Prove Root Cause → Use numbers, not screenshots. Explain every good and bad case.
3. Predict Golden Diffs → Write down exactly which goldens will change before editing code.
4. Fix the Root → Fix the source layer, and delete downstream band-aids.
5. Confirm Invariants → Verify live output, add property tests, check diff count.
6. Update Goldens → Regenerate, revert unintended edits, inspect pixel values.
7. Close the Loop → Commit tests, code, docs, and issue references together.

1. Reproduce and Isolate

- **Shrink the Repro:** Reduce inputs to the absolute minimum (e.g., target width `-w 6` instead of `-w 80`) so the raw ANSI characters can be inspected directly via `cat -v`.
- **Find a Control:** Find a parameter or size that does *not* reproduce the bug. If `-w 6` is broken but `-w 5` is clean, the difference between these two cases is your best clue.

2. Prove Root Cause (Numbers First)

- **Avoid Screenshot-Driven Debugging:** Visual checks confirm *that* something is wrong; they do not explain *why*.
- **Write a Diagnostic Script:** Create a small diagnostic script (e.g. `scripts/debug_probe.go`) that imports the production

functions, feeds the repro inputs, and prints the coordinates and math. Delete this script once done.

- **Formulate the Proof:** You must be able to state the bug as a formula that holds true for every failing case and false for every passing case.

 **Human holds the wheel** — accepting the proof. An agent will happily produce a formula that fits the three cases in front of it; deciding that the explanation is *complete* is a judgement about what has not been tried yet.

3. Predict Golden Impact

- **Compute the Diff Set:** Before updating code, check your golden corpus. List which files will change under your formula.
- **The Red Flag:** If your code change later alters more or fewer goldens than you predicted, stop immediately. Your model of the root cause is incomplete. Revert your code changes and return to Step 2.

4. Fix the Root, Delete the Band-Aids

- **Fix the Origin:** Edit the code at the layer where the wrong calculation originates.
- **Clean the Codebase:** If earlier developers added symptom-suppression hacks downstream, remove them. Leaving downstream band-aids leads to codebase rot.

5. Confirm Invariants

- **Live Check:** Run the repro and verify it matches the expected output.
- **Codify Invariants:** Write unit tests that assert the underlying mathematical property (e.g., “modes must agree on vertical cell snap boundaries”), rather than just comparing strings.
- **Golden Match:** Run the test suite without the update flag. Ensure the failed goldens list is identical to your step 3

prediction.

6. Update Goldens Safely

- Regenerate files using the update flag (e.g., `go test ./cmd/ -update`).
- Use git to checkout and discard any changes to goldens that were not on your prediction list.
- Verify the newly written goldens manually to confirm the pixel adjustments are correct.

 **Agent takes the wheel** — steps 1, 5, and 6: shrinking the repro, writing the invariant tests, and regenerating the predicted goldens. All three have crisp inputs and a mechanical definition of done.

 **The check that makes it safe** — the prediction from step 3 compared against the actual failing-golden list, plus `git diff --stat` on the golden directory. Any file changing that was not predicted stops the change dead; that comparison is what keeps `-update` from ratifying a bug.

Anti-Patterns to Avoid

- **Using `-update` to Make the Suite Green**: Ratifying the bug into the goldens destroys the utility of the test suite.
- **Band-Aid Stacking**: Writing logic to intercept and fix a bug downstream of where it occurred.
- **Reimplementing Code in Diagnostic Scripts**: Always run the *real* production code in your diagnostics.

The Canary That Skipped Itself

Who this is for — anyone who has ever shipped a fix, watched its test go green, and had the bug come back anyway.

Read this if — you write regression tests that shell out to a real external tool.

Skip this if — every check you own runs in-process, with no environment it could be missing.

Takeaways

1. A regression test that can silently skip is not a regression check — it is a check-shaped hole.
2. Don't infer an external tool's version threshold from one installed copy; probe the binary that will actually run.
3. When a bug's only symptom lives inside a GUI the agent cannot see, ask a human to look, one change at a time.

`wayreel` records terminal and desktop demos headlessly; `conreel` runs it inside a container so the same `.reel` script produces the same recording on any host. They are a small, real Solo Fleet pair.

This chapter is one debugging session across both repositories, and it is unusual for this book in one way: **the agent was wrong twice, confidently, before it was right.** Both wrong answers were tested against a real, running binary. Both shipped with a  next to them. That is the point of writing it down.

On Evidence

What is real here. Every command, error message, commit hash, and file path below is taken from the `conreel` and

wayreel repositories and can be checked with `git show <hash>` or by rerunning the command. The dialogue quotes are the actual session transcript, not reconstructed — paraphrasing would have lost the thing worth keeping: how ordinary each wrong turn sounded at the time.

1. A Symptom the Agent Cannot See

`conreel watch reels/gedit-demo.reel` ran clean — no errors, a  `Mirror window opened` line, a normal exit. The human running it said:

“I saw no mpv window”

Nothing in the terminal output disagreed. That is the shape of the whole chapter: **the bug’s only symptom lived inside a GUI window the agent has no way to observe.** Piping `stdout` back and forth cannot answer “is there a window on your screen.”

Tracing the code instead of guessing from the silence found the defect in `wayreel/mirror.go`. `mpv / ffmpeg / wf-recorder` were launched with no `Stdout / Stderr` captured, and the success message printed the instant the *process* launched — not after it had connected to Wayland and mapped a window. A connect failure made the player exit almost immediately, and that failure was invisible: no error text anywhere, the CLI still reported success, the recording continued.

No check existed to catch this, and that absence was the bug: a launched process was being treated as a working one.

The fix captures both processes’ combined output, waits a 300 ms liveness window after launch before declaring success, and prints the captured output whenever the mirror closes for any reason other than the user closing it. The next run surfaced the real error immediately:

```
mpv: ../video/out/x11_common.c:679: vo_x11_init: Assertion
`!vo->x11' failed.
```

`mpv` 's Vulkan/Zink probe was failing in a GPU-less container, and its own fallback chain hit an assertion bug in its X11 backend instead of degrading gracefully. Pinning `mpv` to `--gpu-context=wayland --vo=gpu` plus `LIBGL_ALWAYS_SOFTWARE=1` sidesteps the probe entirely. Verified across all four reels in the repository — `wayreel@7a78ca0`.



Agent takes the wheel — reading `mirror.go`, `scenario.go`, and `main.go` across two repositories to trace exactly what a `👁 Mirror window opened` message does and does not guarantee, then instrumenting the silence.



The check that makes it safe — `conreel watch` `reels/gedit-demo.reel`, which no longer lies. The captured player output is printed on any unexpected close, and the liveness window turns an instant exit into a failure instead of a `👁 Mirror window opened` line.

That part of the session is a clean, standard fix: instrument the silence, read the real error, fix the real cause. The next part is not.

2. Wrong Fix #1: The Threshold Was a Guess

With the mirror fixed, the user was asked to run each reel individually and report what the mirror actually showed — a deliberate protocol, not a formality, because the agent still could not see the window. After one run:

“it was a reel syntax error”

Wrong, but usefully wrong: a specific, checkable claim instead of “something looked off.” A screenshot of the mirror window followed, showing foot’s own startup line:

```
error: foot: /tmp/wayreel-foot.ini:1: [colors-dark]: invalid
section name: colors-dark
```

Not a reel syntax error. A `foot.ini` section name that foot's own parser rejected — on the container's foot 1.25.0, inherited from wayreel's generated theme config.

`FootColorsIni()` (`reelang/types.go`) already had version detection for exactly this, added in an earlier fix (wayreel issue #20): shell out to `foot --version`, use `[colors-dark]` / `[colors-light]` on `foot ≥ 1.23`, and the legacy `[colors]` section below that. The container runs 1.25.0 — past the coded threshold — so it should have worked. It didn't.

The natural next move was to ask what `[colors-dark]` actually *is* to foot, rather than trust the threshold. The container's own shipped default config, `/etc/xdg/foot/foot.ini`, defines only `[colors]` / `[colors2]`, and `foot --check-config` on the container rejected `[colors-dark]` outright — on 1.25.0, a version the existing code believed already supported it.

From that one data point the conclusion looks inescapable: *`[colors-dark]` was never real foot syntax at any version.* The fix that followed dropped the dialect switch entirely and always emitted `[colors]`. It built, it passed `go test ./...`, and a full container rebuild confirmed the error was gone:

“looks clean now”

That confirmation was real. It was also not the end of the story.

3. Wrong Fix #2: The Feature Was Real, Just Newer

The next question — asked by the user, not found by the agent — is the hinge of this chapter:

“The colors issues was created after I saw the host foot complain that ‘colors’ was deprecated and I should use ‘colors-dark’. A warn when foot started. Now you say it never had any ‘colors-dark’ notion?”

That is a direct contradiction of the “never existed” conclusion, from a second, independent foot installation. Checked rather than argued about:

```
$ foot --version
foot version: 1.27.0 -pgo +ime +graphemes +toplevel-tag +blur
-assertions

$ foot --check-config --config=/tmp/test-colors.ini      #
[colors] section
warn: config.c:1635: /tmp/test-colors.ini:2: [colors]:
deprecated; use [colors-dark] instead

$ foot --check-config --config=/tmp/test-colors-dark.ini  #
[colors-dark] section
$
# accepted, no warning
```

Both installations were telling the truth. The container’s foot 1.25.0 genuinely rejects `[colors-dark]`. The host’s foot 1.27.0 genuinely deprecates `[colors]` in favor of it. Foot’s own changelog gives the boundary: **1.26.0**, released 2026-03-03, renamed `[colors]` / `[colors2]` to `[colors-dark]` / `[colors-light]`.

So the feature was never fictional. The original threshold — 1.23 — was simply the wrong number, and “always emit `[colors]`” was a second wrong fix that would have quietly reintroduced, on any newer foot, the exact defect issue #20 existed to remove: a deprecation warning baked into every recording.



Human holds the wheel — noticing that a confident, tested, “corrected” conclusion contradicted something they had personally seen on their own machine, and saying so instead of accepting the correction. Neither foot version was in the agent’s own environment; only a human checking a second real machine surfaced the contradiction.

4. Why Both Wrong Fixes Looked Verified

Both conclusions shipped with a  beside them: 1.23 in `wayreel/issues/20-foot-ini-colors-dark-compat.md`, “never real syntax” in `conreel/issues/05-foot-colors-dark-workaround.md`. And `reelang_test.go` already had `TestFootColorsIniAcceptedByFoot` — a real test, running a real `foot --check-config` against the generated config, exactly the kind of check [Canary-First Development](#) argues for. It should have caught both mistakes on the first `go test ./...`.

It didn’t, because of one line:

```
func TestFootColorsIniAcceptedByFoot(t *testing.T) {  
    if _, err := exec.LookPath("foot"); err != nil {  
        t.Skip("foot is not installed")  
    }  
    // ...  
}
```

That guard is reasonable in isolation — don’t fail CI on a machine without `foot`. A second defect compounded it. A sibling test, `TestFootColorsIniLegacyDialect`, forced the legacy code path for its own assertions by calling `footVersionOnce.Do(func(){})` — a `sync.Once`, spent, for the rest of that test binary’s life. Any *later* test wanting a genuine, fresh probe silently got the forced value instead. `TestFootColorsIniAcceptedByFoot` ran, didn’t skip, and reported green — while re-verifying whatever the previous test had forced, not the detection logic at all.

Two silent failures, stacked: a check that can no-op without saying so, and shared process state that one test could poison for every test after it. Both are easy to write by accident, and both produce exactly the same outward signal as a check that works — a green suite.

5. The Fix That Held: Stop Guessing, Start Probing

The threshold was wrong once at `1.23` and would be wrong again at any other hardcoded number the next time `foot` moves this boundary. `FootColorsIni()` now probes the *installed* `foot` binary at call time instead of encoding a version anywhere:

```
func detectFootColorThemeSupport() {
    f, err := os.CreateTemp("", "wayreel-foot-probe-*.ini")
    if err != nil {
        return
    }
    defer os.Remove(f.Name())
    if _, err := f.WriteString("[colors-dark]\n"); err != nil
{
        f.Close()
        return
    }
    f.Close()

    out, err := exec.Command("foot", "--check-config", "--
config="+f.Name()).CombinedOutput()
    footSupportsColorThemes = err == nil &&
!strings.Contains(string(out), "invalid section name")
}
```

Cached once per process, and defaulting to the legacy `[colors]` dialect — not the modern one — when `foot` is missing or the probe fails. `[colors]` stays valid, if deprecated, on every `foot` version: guessing the *new* dialect wrong breaks outright, guessing the *old* one wrong only warns.

The test fix mattered as much as the code fix. The poisoned `sync.Once` is now reset to a fresh state on every forced-dialect test's cleanup, and `TestFootColorsIniAcceptedByFoot` resets it at its own start too, so it can never again inherit another test's forced value.



Agent takes the wheel — writing the probe, rewriting the tests so a forced dialect can no longer poison a later real

one, and rebuilding the container to exercise both foot 1.25.0 (legacy fallback) and foot 1.27.0 (modern dialect).

 **The check that makes it safe** — `go test ./reelang/`, where `TestFootColorsIniAcceptedByFoot` now runs a genuine probe every time it does not skip, plus a live `conreel watch reels/smoke.reel` on both foot versions, confirmed by the human watching the window (`wayreel@7a78ca0`, `conreel@00db655`).

6. The Rule This Session Earned

Neither wrong fix was a careless guess. Each was checked against a real, running `foot` binary, on a real machine, and confirmed clean. The mistake both times was treating *one* real environment as *the* real environment — and the regression test built to catch exactly that had been sitting there the whole time, never getting a chance to run honestly.

So the rule is not “test it.” It is this line, now in `wayreel/AGENTS.md`, outside the section `harnes` manages so it survives the next tooling sync:

Before marking an issue Resolved that depends on an external tool’s version-gated behavior, run its canary/regression test and confirm it didn’t skip — a skipped check is not a verified fix.

The companion habit is harder to write down, because no file can enforce it: when the only evidence of a bug is something a human can see and the agent cannot, the fastest path is not a better guess from the transcript. It is asking, one change at a time — and treating “looks clean now” as real evidence right up until a second, independent check says otherwise.

The same mistake wearing different clothes — one machine’s package set taken for every machine’s — is in [The Single-Prompt Miracle and What It Cost](#) \$4, where a correct fail-closed fix made a

headline feature unreachable on every distribution that does not ship `pdftk`.

Full history, including both wrong “Resolved” notes left in place rather than rewritten away, is in `wayreel/issues/20-foot-ini-colors-dark-compatible.md` and `conreel/issues/05-foot-colors-dark-workaround.md`.

The Solo Fleet, Reviewed

Where CLI-First & Local-First Win — and Where They Break

Who this is for — engineering leaders weighing which of these practices survive contact with an organization.

Read this if — you suspect a solo methodology cannot transfer.

Skip this if — you are here for the terminal craft; that is [TUI Restoration](#).

Takeaways

1. The failures of this approach at scale are failures of *scope*, not of substance.
2. Verification before creation is the one idea that transfers unchanged to any team, stack, or process.
3. CLI-first and local-first are an engineering substrate, not a company operating model.

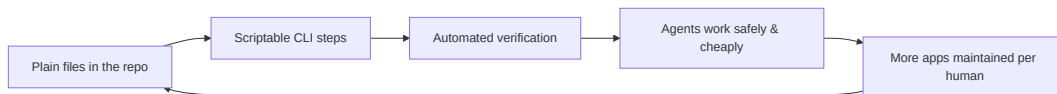
Every book like this one has a blind spot: it was written inside the world it describes. This whole playbook assumes one developer, full control, a terminal, and a repository that answers to nobody else. That assumption deserves an honest review — not a victory lap.

This chapter turns the lens around. First, *why* the CLI-first, local-first approach works so well here. Then, candidly, where it strains or outright fails once teams, meetings, products, and money enter the room. And finally — because the answer is not “so do everything in Jira” — what those organizations should still learn and adopt from this way of working.

Why It Works Here: The Solo Leverage Loop

For a single developer maintaining many apps, CLI-first and local-first are not aesthetic preferences. They are the only economics that work:

- **Zero coordination cost.** Every decision is a file edit away. No tickets, no sync meetings, no waiting for a dashboard admin. The entire “process” is `git log`.
- **Everything is scriptable, so everything is verifiable.** A CLI tool composes into a canary, a Makefile target, a preflight gate. A GUI workflow composes into a wiki page titled “How to click through the deploy” that is wrong within a month.
- **Context lives in the repo, so agents are first-class citizens.** `AGENTS.md`, `spec/`, `issues/`, `CONTEXT.md` — an AI agent (or the developer, after six months away) reads the same files and gets the same truth. There is no tribal knowledge because there is no tribe.
- **Offline-first means forever-first.** No CDN rot, no SaaS sunset, no login wall between you and your own tooling. The build of 2026 will still run in 2036.



The loop compounds: every app added to the fleet makes the conventions more valuable, and the conventions make every next app cheaper. That is the engine of the solo fleet.

The Honest Part: Where It Strains and Fails

Take the same practices into a ten-person team, a product organization, or a revenue meeting, and several of them stop being strengths. Not because the ideas are wrong — because their *preconditions* are gone.

1. The repo stops being where decisions happen

Local-first says: “If it is not in the repository, it does not exist.” In an organization, most decisions that matter — pricing, priorities, headcount, a customer escalation — happen in meetings, chats, and slide reviews. **The repo is downstream of the real decision flow.** Insisting that Markdown files are the single source of truth makes the engineering team’s truth invisible to everyone who cannot (or will not) read a Git repository: product managers, designers, sales, leadership.

2. Coordination does not scale through files alone

One person needs no locking protocol. Ten people editing `issues/` as flat Markdown recreate — badly — what issue trackers solved long ago: assignment, notification, status, deduplication, cross-team visibility. The solo fleet’s greatest feature (*no process*) is precisely what a team cannot afford. Files don’t page anyone at 3 a.m., and `git blame` is not an incident-response rotation.

3. The bus factor is a feature flag you can’t ship

A fleet tuned by one person is exquisitely shaped to that one person’s judgment. The conventions are documented, yes — but the *taste* that decides when to break them is not. Companies rightly fear this: a system that only its author can safely evolve is a liability with excellent test coverage.

4. “Human in control” assumes the human owns the outcome

Manually triggered skills and hand-steered agents are ideal when the operator is also the architect and the owner. In a company, the person triggering the pipeline is often *not* the person accountable for the result. Organizations respond with the exact things this book warns against — approval layers, audit trails, platform teams, managed harnesses — and they are not wrong to. **Governance is what “human in control” looks like when there are four hundred humans.**

5. Money measures outcomes, not craft

The golden-change playbook's discipline — prove the root cause before touching a pixel — is engineering at its best. But a product deadline does not care that your model of the bug is incomplete; sometimes shipping the band-aid *is* the rational business decision, with a ticket to fix the root later. A methodology with no vocabulary for “acceptable debt under deadline” reads, from a P&L, like perfectionism with a Makefile.

6. Meetings are the API of organizations

This playbook has no slide about meetings because the solo fleet has none. But an organization's alignment layer *is* synchronous: demos, reviews, decisions in rooms. A local-first purist who communicates only through beautifully versioned Markdown is, organizationally, offline. (This repository ships slide decks for a reason.)

Practice	Solo fleet	Team / org / product reality
Files as single source of truth	Perfect memory, agent-ready	Excludes non-repo stakeholders; truth forks into chats & decks
No process, no tickets	Zero overhead	No assignment, paging, or cross-team visibility
Manually triggered agents	Full control, full accountability	Doesn't scale past the architect; becomes a bottleneck
Root-cause-only fixes	Compounding quality	Sometimes loses to deadline economics — and should
Offline-first everything	Forever-durable	Fights SSO, compliance, managed platforms — often for good reasons

What Companies Should Steal Anyway

The failure modes above are real, but they are failures of *scope*, not of substance. Strip the purism and the core practices transfer beautifully — most organizations are drowning in exactly the problems these practices solve.

1. Verification before creation — as policy. The single most transferable idea in this book. Before a team adopts AI coding agents, fund the harness: preflight gates, canaries, golden suites. An organization that lets agents (or contractors, or juniors) produce code faster than it can verify it has automated the creation of review debt. This principle needs no terminal romance; it works in any stack, any process.

2. Decision records where the work lives. Companies don't need to abandon their trackers — they need to stop losing *why*. Lightweight ADRs and a `CONTEXT.md` glossary inside each repo cost minutes and end the archaeology of “why is this service like this?” — for new hires and AI agents alike. Meeting decisions that affect code should get a two-paragraph file in the repo. The meeting can stay; the amnesia goes.

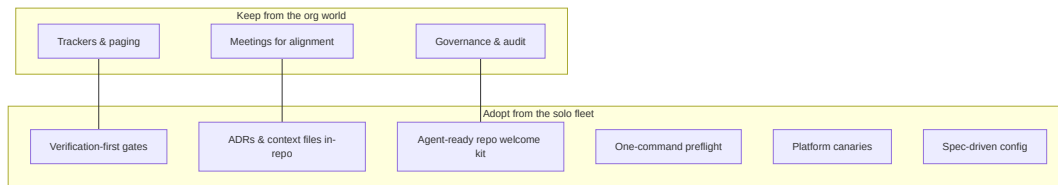
3. Agent-readable repositories as an onboarding standard. The solo fleet's superpower is that any repo greets a cold visitor with `AGENTS.md`, a spec folder, and one verify command. That is not a solo trick — that is the best onboarding document ever written, and it doubles as AI-agent infrastructure. Companies that standardize a “repo welcome kit” get faster onboarding *and* safer agent adoption from the same artifact.

4. One-command truth: `make preflight` culture. Every repo, one command, green or red, runnable locally and in CI, no dashboard login required. Where builds depend on a bespoke cloud console, debugging becomes a permissions request. Keeping the *critical path* scriptable and local-runnable is cheap insurance — even when the org platform runs the official pipeline.

5. Canaries for the platform, not just the laptop. Teams build on internal platforms the way this book builds on Chromium: full of unverified assumptions. A thirty-line canary that proves “the staging bucket is writable, the auth token refreshes, the sandbox

allows this syscall” — checked into the repo and run in CI — kills whole classes of cross-team debugging tickets.

6. Spec-driven config over tribal constants. Fonts, colors, limits, and flags in one schema-validated spec file, with derivatives generated — this chapter’s own book styling works that way. In a company, that pattern is the difference between “ask Dave which value is real” and a diff anyone can review.



The Closing Reflection

The honest summary is this: **CLI-first and local-first are not a company operating model — they are an engineering substrate.** They optimize the layer where code is made and verified, and at that layer they win everywhere, from solo fleets to enterprises. The mistake would be reading this book as an argument against process, product thinking, or meetings. It is an argument that *beneath* all of those, every repository should be self-explaining, self-verifying, and operable by any collaborator — human or agent — with nothing but a checkout.

Organizations that adopt that substrate keep their trackers, their governance, and their roadmaps — and gain what the solo fleet has: repositories that never forget, checks that never sleep, and agents that can finally be trusted with real work.

And if one developer with a terminal can keep a dozen apps alive, imagine what a team could keep alive — once its repos can explain themselves.

Acknowledgments

The biggest thank-you belongs to my wife, **Anja**. She gives me all the time to build these tools, write these books, and keep educating myself — evenings, weekends, and every “just one more fix” in between. None of this would exist without her patience, encouragement, and love. This book is hers as much as mine. ❤️

A warm shoutout also goes to **Upvest**, my previous employer, who gave me the chance to work with AI developer tooling from the very beginning — and to use it in real production environments, where many of the lessons in this book were first put to the test.

Colophon

Who this is for — readers who want to know how a book about agentic development was itself made.

Read this if — you are curious what the split of labour actually looked like.

Skip this if — you came for the engineering; the [Golden Path](#) has more of it.

Takeaways

1. The books are built by the same pipeline they describe: spec, canary, harness, preflight.
2. Attribution is machine-readable — model credit lives in commit trailers, not in prose.
3. The agents got things wrong here too, and the wrong things are named below.

Last updated: 2026-08-09.

How This Book Was Made

Everything you are reading is plain Markdown in `src/`, compiled by [mdBook](#) and printed to PDF by headless Chromium. The slide decks that accompany the library are compiled by a single-file Go program, `scripts/build_slides.go`, which also generates each book's stylesheet from `spec/config.yaml` and refuses to build if a forbidden term appears in any source file. One command builds the shelf; one command, `make preflight`, decides whether it was allowed to.

The typography, accent colors, margins, and slide transitions of all three books are spec values, not stylesheet values. Changing the violet you see in these headings is a one-line edit in `spec/config.yaml` — the mechanism, and the bug it shipped with, is the subject of [The Golden Path](#).

Tool	Version	Role
mdBook	v0.5.3	compiles Markdown into the books
Chromium	headless	prints books and decks to PDF
Go	v1.22	slide compiler, dev server, test harness
Zig	v0.16.0	the language of <code>emojig</code> , the case study

(Versions as recorded in the `tools:` registry of `spec/config.yaml`.)

Who Wrote What

The human — Uwe Jügel — wrote the thesis, the structure, the arguments, the decisions, and every judgement call about what belongs in the spec, what belongs in a rule, and what belongs in an ADR. The models drafted, refactored, wrote tests, and did the mechanical work of turning notes and Git history into prose.

Attribution is recorded where it can be queried rather than claimed. At the time of writing, 16 of this repository's 37 commits carry a `Co-Authored-By:` trailer naming the model that worked on them; the earlier commits predate the convention. Run the count yourself:

```
git log --format='%H %(trailers:key=Co-authored-by,valueonly)' | awk 'NF>1' | wc -l
git rev-list --count HEAD
```

Which models may be named at all is itself governed by the `models:` registry in `spec/config.yaml` — the build aborts on any model name that is not in it, which is why this page will not quietly drift into naming something that never worked here. The registry records **Claude Fable 5** for technical writing and general agentic work, **Claude Sonnet 5** for the module refactors and the final book compilation, **Claude Opus 4.8** for the image pipelines, and **Claude Sonnet 4.6**, **Gemini 3.1 Pro**, and **Gemini 3.5 Flash** for the `emojig` TUI work chronicled in the [Emojig Chronology](#).

What the Agents Got Wrong

A colophon in a book with this thesis owes the reader the failures, not just the credits. Three, all documented elsewhere in the repository:

1. **A one-line YAML parse that was right for every case but the real one.** `strings.Trim(..., "\"'")` handled quoted values perfectly and mangled every value with a trailing comment — including the accent color of every book. It passed review, passed the build, and shipped. Post-mortem and fix: [The Golden Path](#).
2. **Two correct stylesheets that disagreed about authority.** A hand-written theme and a spec-generated baseline both defined the same selectors; the book rendered as a colour hybrid with every test green. Resolved by a decision record rather than by code.
3. **A review that critiqued a table of contents.** An AI reviewer recommended adding diagrams to the two chapters that already had them, and missed the four chapters that had none. It is a useful reminder of what agent output looks like when nothing in the loop forces contact with the source: fluent, confident, and about a book that does not exist.

The first two were caught by a human noticing that a page looked wrong, then converted into permanent checks. The third was caught by counting. All three are the same lesson from different angles: **an agent's output is a hypothesis, and the only thing that makes it knowledge is a check.**

Acknowledgments

The dedications belong at the end of [The Solo Fleet, Reviewed](#), where they were written, not here.

How to Read the House Rules

Who this is for — anyone about to write or review `Go`, `Zig`, `Bash`, or `Make` in these repositories, human or agent.

Read this if — you want to know *why* the subset is drawn where it is.

Skip this if — you just need the rules; go straight to [Go](#), [Zig](#), [Bash](#), or [Make](#).

Takeaways

1. These four pages are reference material, not narrative. They are meant to be grepped, not read.
2. The subset is chosen for one property: whether a reviewer — or a check — can tell right from wrong quickly.
3. A rule that no command can enforce is a preference. It is allowed here, but it is labelled as such by living in prose rather than in `make preflight`.

Why a Subset At All

“The Good Parts” is Crockford’s phrase, and borrowing it sets an expectation: that somebody argued for a subset, rather than merely listing a house style. So here is the argument, in one paragraph, before the lists start.

Every language feature has a cost that is invisible when you write it and obvious when someone else reads it. In a fleet maintained by one person and a rotating cast of models with no memory, *reading* is the dominant cost. A package-level mutable variable in `Go` costs nothing to type and costs a future session an hour of tracing hidden coupling. A `[[ ]]` test in `Bash` works perfectly until the script runs under `sh` on a machine you did not configure. A

clever `@compileError`-driven abstraction in Zig is a delight to write and an obstacle to every subsequent change.

So the subset is drawn by one criterion, and it is not elegance:

Keep the constructs whose correctness a cold reader — or a check — can establish locally. Drop the constructs that require holding the rest of the program in your head.

That criterion explains every rule in the four pages that follow. Explicit allocators in Zig, explicit dependencies in Go, `test` instead of `[[ ]]` in Bash, self-documenting targets in Make — each one trades a little writing convenience for a large amount of reading certainty. It is the same trade the rest of this book makes everywhere else: pay up front, in the harness, so that nobody has to remember anything later.

What These Pages Are Not

They are not tutorials, and they are not comprehensive. They are the accumulated residue of real bugs in `emojig`, `cati`, and this repository — the rules that were paid for. Where a rule is pinned to a specific toolchain version, it says so at the top of the page; see [Zig — House Rules](#), which is the most perishable of the four.

Enforcement

Rule class	Where it lives	How it is enforced
Formatting	<code>gofmt</code> , <code>zig fmt</code>	<code>make</code> <code>preflight</code> fails on unformatted Go
Structure & dependencies	these pages	human and agent

Rule class	Where it lives	How it is enforced
		review
Behaviour	<code>scripts/build_slides_test.go</code>	<code>make test</code>
Environment	<code>scripts/canary.sh</code>	<code>make canary</code>

The second row is the honest one: those rules are enforced by attention, not by a command. That is exactly why they are written down in one place a cold session can read in thirty seconds.



Go — House Rules

Who this is for — anyone writing or reviewing Go in these repositories. Reference material: grep it, don't read it.

Read this if — you are about to add a dependency, a global, or a test framework.

Skip this if — you want the reasoning; it is in [How to Read the House Rules](#).

Takeaways

1. Standard library first; every dependency needs a reason and approval.
2. No package-level mutable state and no `panic()` outside boot.
3. Table-driven tests in the same package, standard `testing` only.

Go is a language designed for simplicity, readability, and speed. To maintain these qualities in agent-assisted development, we adhere to the following subset and conventions.

Language & Dependencies

- **Modern Go:** Leverage current features like `any` and generics, but only where they explicitly reduce boilerplate/noise.
- **Minimal Dependencies:** Default to the Go Standard Library. If you need a library, get user approval first.
- **Permitted CLI Library:** Use `github.com/spf13/cobra` for CLI-based tools.
- **No Heavy Frameworks:** No ORMs (write raw SQL), no logging frameworks (use standard `log`), no Dependency Injection (DI) containers (pass dependencies explicitly).

State Management

- **No package-level mutable variables:** Pass state explicitly via function arguments or state structs (e.g. `type App struct { client *http.Client }`). Mutable global state creates hidden coupling and breaks tests.
- **Safe `init()` usage:** Use `init()` only for static, side-effect-free registrations. Never connect to databases, services, or load files during initialization.

Spec-Driven Development

- Avoid hard-coding application configuration, screen texts, labels, or layout variables.
- Define them in YAML specs under `spec/` (e.g., `spec/strings.yaml`, `spec/layout.yaml`) and embed them in the binary using `//go:embed`.
- Write compiler/unit test assertions to verify that your Go structs match the specs.
- The pattern in practice: [Week 2 of the Emojig Chronology](#) moved a TUI's entire configuration out of code, and [The Golden Path](#) walks the same move — and its first bug — in this repository's Go compiler.

Error Handling

- **Wrap errors with context:** Use `fmt.Errorf("module: %w", err)` to preserve the original error while giving readable context.
- **No panic:** Never use `panic()` except for truly unrecoverable boot-time configurations.
- **Return errors:** Let errors bubble up to `main()` where they are printed and exited.

Output and Formatting

- Functions should return values, not print them directly (unless they are explicitly printer helpers).
- Indent output messages for visual hierarchy:
 - Print changed items with two spaces: `wrote README.md`
 - Print sub-details with four spaces: `size: 1.2 KB`

Unit Tests

- Use standard Go table-driven tests (`t.Run`).
- Avoid external testing libraries, assertions frameworks, or mock generators. Standard library `testing` is sufficient.
- Put tests in the same package (e.g. `package main` in a test file alongside the main file) to test unexported components easily.

Zig — House Rules

 **Version banner: Zig 0.16.0, verified 2026-07. These APIs move.** The conventions in the first section are stable house rules. The API shapes in the second section (`pipe2` , `StdIo.file` flags, `Permissions.fromMode` , peer-type resolution) are notes from a moving target — they were true against 0.16.0 and the compiler, not this page, is the authority. If your compiler disagrees, your compiler is right.

Who this is for — anyone writing Zig in `emojig` or a sibling project. Reference material: grep it, don't read it.

Read this if — you hit a 0.16.0 API that does not look like the docs.

Skip this if — you want the reasoning behind the subset; that is [How to Read the House Rules](#).

Takeaways

1. Allocations are always explicit; hot paths allocate nothing.
2. Utility binaries ship `-O ReleaseSmall` , under 1 MB, under 5 MB RSS.
3. Treat every pinned API shape below as dated evidence, not as doctrine.

Zig is a general-purpose programming language designed for robustness, optimality, and clarity. To maintain these qualities in agent-assisted development, we adhere to the following conventions, plus the Zig 0.16.0 API shapes discovered during project development.

Language & Allocation Conventions

- **Explicit Allocation:** Never hide allocations. Pass an `Allocator` explicitly to functions that need to allocate memory.
 - **Zero-Allocation Cores:** For hot paths (such as fuzzy search engines or rendering loops), design a zero-allocation core. Prefetch data or use stack-allocated buffers.
 - **ReleaseSmall Optimization:** For utility binaries, optimize using `-O ReleaseSmall`. A typical statically linked binary (e.g., `emojig`) should be under 1 MB and use less than 5 MB of RSS RAM.
 - **Clean Namespace Separation:** Avoid monolithic root files. Move helper logic to dedicated files and use `std.testing.refAllDecls(@This())` in your main test blocks to run tests in separated modules.
-

Zig 0.16.0 API Pitfalls & Patterns

Dated notes — verified against Zig 0.16.0 in July 2026. Every one of these was paid for by a compile error during [emojig's build-up](#), where early 0.16.0 type mismatches caused most of the week's failed builds.

1. Pipes — Use `std.os.linux.pipe2` on Linux

In Zig 0.16.0, `std.posix` does not expose `pipe2`. Use the Linux-specific syscall wrapper for non-blocking execution:

```
var fds: [2]std.posix.fd_t = undefined;
const flags = std.os.linux.O{ .NONBLOCK = true, .CLOEXEC =
true };
const rc = std.os.linux.pipe2(&fds, flags);
switch (std.posix.errno(rc)) {
    .SUCCESS => {},
    else => return error.SystemResources,
}
```

2. Spawning Subprocesses with Stdout Redirection

When redirecting standard I/O to a file descriptor using `StdIo.file`, Zig 0.16.0 requires providing a `flags` field:

```
var child = try std.process.spawn(io, .{
    .argv = &argv,
    .stdin = .ignore,
    .stdout = .{ .file = .{
        .handle = pipe_fds[1],
        .flags = .{ .nonblocking = false },
    }},
    .stderr = .ignore,
});
```

Note: Close the write end of the pipe in the parent process immediately after spawning, or the read end will block forever on EOF.

3. File Permissions via Permissions Enum

Do not use raw numeric modes (e.g. `0o600`) when creating or modifying files in Zig 0.16.0. Use the `std.fs` types wrapper:

```
const file = try std.fs.createFileAbsolute(io, path, .{
    .permissions = std.fs.File.Permissions.fromMode(0o600),
});
```

4. Slice vs. Array Pointer Peer-Type Resolution

Constant string literals (e.g., `""`) are of type `*const [0:0]u8`. Standard library formatting tools (like `std.fmt.bufPrint`) return mutable slices (`[]u8`). Peer-type resolution will fail if both are evaluated in a simple `catch` expression:

```
// ❌ Compile error: incompatible types
const s = std.fmt.bufPrint(&buf, "{s}", .{val}) catch "";

// ✓ Fix: Use an if-expression to yield identical types
const s = if (std.fmt.bufPrint(&buf, "{s}", .{val})) |res|
res else |_| buf[0..0];
```

5. std.mem.trim Const/Mutable Safety

`std.mem.trim` always returns `[]const u8`. If you are trimming a mutable buffer and require a mutable slice as a result, wrap the returned slice in `@constCast`:

```
const trimmed = std.mem.trim(u8, buf[0..len], " \t\n\r");
return @constCast(trimmed); // Safe: the underlying buffer is
mutable
```

Bash — House Rules

Who this is for — anyone writing a canary, a build script, or any glue in these repositories. Reference material: grep it, don't read it.

Read this if — you are about to type `[]`.

Skip this if — you want the reasoning behind the subset; that is [How to Read the House Rules](#).

Takeaways

1. `set -euo pipefail`, always, on line two.
2. `if test ...` — never `[ ... ]`, never `[[ ... ]]`, and `then / else / do` on their own lines.
3. Quote every expansion; assume the default `awk` is `mawk`.

Bash is an incredibly powerful glue language when restricted to its robust, readable features. Forget old habits and legacy scripts. This is the subset we write.

1. The Strict Header

Always start script execution with:

```
#!/usr/bin/env bash
set -euo pipefail
```

- `-e`: Exit immediately if any command returns a non-zero status.
- `-u`: Exit if an uninitialized variable is referenced.
- `-o pipefail`: Ensure that pipeline return codes reflect the last non-zero command in the chain, not just the final command.

2. Conditionals: Use test Command

Never use the brackets `[ ... ]` or `[[ ... ]]`. Instead, use the clean and standard `test` command.

```
if test -f "$file"
then printf 'Found %s\n' "$file"
fi

if test "$a" = "$b"
then printf 'Equal\n'
else printf 'Not equal\n'
fi
```

- Put `then` / `else` / `do` always on their own lines.
- Avoid semicolons where possible.

3. Variable Safety

- Always double-quote variables when accessing: `"$var"` or `"${var}"`.
- Handle required arguments with default error patterns:
`pattern="${1:?Usage: script.sh PATTERN}"`
- Use `local` inside functions. For command substitutions, declare first, then assign, to prevent `local` from masking execution exit codes:

```
local result
result=$(command args)
```

4. Alignment Continuation

Do not follow rigid N-space indents. Instead, use alignment continuation to make long command pipelines and logic structures readable:

```
# Line up commands under the start of the chain
result=$(some_command |
    grep "pattern" |
    awk '{print $2}')

# Line up conditions under the first test
if test -f "$a" &&
    test -d "$b"
then stmt1
    stmt2
fi
```

5. Output discipline

- Use `printf` instead of `echo` for printing variables: `printf '%s\n' "$var"`.
- Write helper functions to handle pass/fail messages and stderr logging:

```
pass() {
    printf '  ✓ %s\n' "$*"
}

fail() {
    printf 'ERROR: %s\n' "$*" >&2
    exit 1
}
```

This is the shape every script under `scripts/` uses — the page and the repository agree, which is the only reason to trust either.

6. Awk Portability (Warning)

On standard Debian, Ubuntu, and Raspberry Pi OS, the default `awk` is **mawk**, not gawk. Avoid gawk extensions:

-  Do not use 3-argument `match()`.

-  Do not use `strtonum()` .
-  Do not use `gensub()` . Use standard portable constructs like `split()` , `sub()` , and `gsub()` .

Make — House Rules

Who this is for — anyone adding a target to a project Makefile. Reference material: grep it, don't read it.

Read this if — you need the  sentinel or the self-documenting `help` recipe.

Skip this if — you want the reasoning behind the subset; that is [How to Read the House Rules](#).

Takeaways

1. `help` is the first target; bare `make` prints usage.
2. The  sentinel replaces per-target `.PHONY` bookkeeping.
3. Express order as dependencies, and always run the locally built binary.

GNU Make is the ultimate tool for orchestrating local-first development workflows. By enforcing a clean structure and a few modern conventions, Makefiles stay maintainable and self-documenting.

1. Structure & Defaults

- The first target in the Makefile must always be `help`. Running `make` alone should display usage instructions.
- Add a blank line between targets to separate them visually.

2. Variables

- Use `:=` for immediate variable assignment.
- Use `?=` for variables that should be overridable by environment variables (e.g. `PREFIX ?= /usr/local`).
- Align `=` signs for clean reading:

```
BINARY    := books
BUILD     := build
PREFIX    ?= /usr/local
```

3. The Sentinel Trick

Declaring `.PHONY` rules manually for every target is tedious and prone to omissions. We use the Unicode gear  sentinel to implicitly make all targets phony:

```
.PHONY:  # make all commands phony
```

Adding  as a dependency to every target ensures that it always runs:

```
build:  # compile the application
    go build -o $(BINARY) .
```

4. Self-Documenting Help

We scrape comments starting with  and description comments (`#` or `##`) to build a dynamic help target:

```
help:  ## show this help
    @grep -E '^[a-zA-Z_-]+:.*?.*#+ ' $(MAKEFILE_LIST) | \
    awk 'BEGIN {FS = ".*?.*#+ ";} {printf "  %-15s %s\n", $$1, $$2}'
```

Any target defined with `name:  ## description` — or with a single `#` — will be automatically formatted and listed when the user types `make`.

5. Dependency Chains

Ensure action targets build the application first:

```
build: ⚙️ ## build target
      go build -o $(BINARY)

run: ⚙️ build ## run the locally-built binary
     ./$(BINARY)
```

Always invoke the locally-built binary (e.g., `./$(BINARY)`), never the global one on the `$PATH`.