First-Class

Building for humans and agents alike

ZACH "LEIF" ERIKSEN

First-Class

Building for humans and agents alike

ZACH "LEIF" ERIKSEN

Copyright

Free to read online. The ePub is pay-what-you-want; if it helped you, you can support the work.

github.com/0xLeif · leif.algo

One of four books in the agent-stack set. How it was made is in the colophon at the back.

Dedication

For everyone who builds in the open, and ships it anyway.

The Library

These books stand alone, but they were written as a set. Code got cheap and trust got scarce. Together they are one argument: what to build now, and how to trust it.

- **The Agent Developer's Field Guide**: Building tools, specs, and trust for agents that ship real code
- **First-Class**: Building for humans and agents alike *(this book)*
- **Building Agents**: Notes from trying to give software its own hands
- **Open Source Tooling**: Building tools people actually use

Free to read online. Each ePub is pay-what-you-want.

Contents

Introduction

This is the short, stubborn book of the four, and the one the others are really arguing for.

Most tools are built for humans, and then an agent gets bolted on the side: a second API, a flag, a special mode that does half of what the real product does. I think that is backwards. The claim here is simple. A tool should be first-class for both. A human should be able to drive it without an agent, and an agent should be able to drive it without a human, across the same surface, with no second-class door.

I came to this from building, not from theory. I make small Swift libraries and developer tools, and the ones that aged well were the ones where the core was clean enough that anything could call it. When agents showed up, those tools were already ready. The ones that were not are the ones I keep apologizing for.

If the *Field Guide* is the method, and *Building Agents* and *Open Source Tooling* are the evidence, this is the thesis they all serve. You do not need the other three to use this one. It is here to change how you look at the next tool you build or pick: ask whether it treats a human and an agent as the same kind of citizen, and watch how much follows from the answer.

It is short on purpose. Read it in one sitting.

Both citizens

A lot of my tooling comes from one idea: humans and agents are going to use the same tools.

So here's the definition the whole book runs on, up front and plain: first-class means a human drives the tool alone, an agent drives the tool alone, the same commands both ways. Not two products. One tool, two kinds of user, neither one the special case the other gets translated into.

When I say *first-class*, I mean it the way a programmer means it: a real, supported participant the tool was actually built for, not a guest it tolerates. Not a flight upgrade. A first-class user is one the tool was designed around, with a real way in. Not one that has to be translated into somebody else's shape first.

There can be agent-first tools, built only for agents. There can be human-first tools, built only for people. The interesting case, the one almost nobody is building for, is the tool that's both at once. That's what I mean by *first-class for both*, and what the rest of the book means too. It's a stricter thing than agent-first: agent-first only has to serve the agent, while first-class-for-both has to serve a person and an agent across the same surface without either side getting a worse version. Right now we mostly have human-first projects and we're bringing agents *into* them. The agent shows up late, to a tool that was built for somebody else, and we hope it figures the thing out.

What we actually need is human-and-agent-first projects. Built that way from the start.

"Human-first with agents bolted on" is the default because it's the path of least resistance. You already have the tool, it already works for you, and when an agent comes along the easy move is to wrap a little glue around the thing you've got. It works, kind of. You took a tool that was designed around a human's eyes and a human's hands and you asked a process with neither to pretend it had both. What that costs an agent (the hang on a prompt, the output it can't read, the relearning every run) gets its own chapter later. Here it's enough to say the agent ends up papering over a gap on every line that assumed a human.

So flip the assumption. Build the tool so it works first-class either way. That's the definition from the top, restated as a build instruction: you don't have a real product and an "API mode" stapled on the side, and you don't have a CLI and a separate agent shim that drifts out of sync the first time you change anything. Both are real users. Both are supposed to be there.

And here's the part that matters most to me. When the agent is a first-class user, the tool can actually *help* it, instead of making it guess at all the commands and how things work. A human can muddle through a confusing tool. They'll read the README, try a thing, read the error, try

another thing, ask a coworker. An agent muddling through is just expensive guessing. A tool that was built for the agent tells it what's possible, hands it output it can use directly, and fails in a way that says what to do next.

This is the same tool humans want. Discoverable commands, output you can trust, errors that tell you what went wrong. The agent just can't paper over the gaps the way a human can, so building for the agent forces you to close them. Designing for both makes the software better.

People hear "build for agents too" and assume it means two products, or a compromise where each side gets a worse version of what it wanted. It's one good core with a step where you expose it to both. That's not double the work, and the next chapter shows why.

The reason I keep coming back to this is that the agents are only going to do more over time, not less. Today they fumble through human tools. Tomorrow they're doing a real share of the work, and the share goes up. If the tools we build now assume a human is always there to handle the prompt, read the screen, click the button, we're building a future on top of an assumption that's getting falser every month.

There's a bigger version of this argument too. We've had a decade of code written entirely by humans, and now AI is writing code on top of all of it, and at some point that mix tips. The tools are what decide whether that goes well or badly. That's a whole chapter on its own: the bridge.

For now the claim is just this: humans and agents will use the same tools. So build them for both, as equal first-class citizens, from the start.

First-class for an agent

So what does a tool actually need for an agent to be a first-class user, and not a human tool an agent fumbles through?

Four things. None of them exotic.

Structured, machine-readable output. A real serialization format, JSON is the one I reach for, so the agent gets data, not a screen. Most tools hand back pretty text: aligned columns, color, a summary line at the bottom. That's for a human's eyes. An agent has to scrape it, and the scraping breaks the day you change the spacing. Give it the actual data. It reads a field instead of parsing a paragraph.

Discoverable, consistent commands. Consistent verbs across the tool, and a self-describing `--help` the agent can read to find out what's possible. If the help text is real, the agent reads it and knows what the tool can do. It doesn't have to have seen this tool before. It asks the tool, and the tool answers.

Errors that guide the next step. When something fails, say what to do about it. Not a stack trace, not `error: 1`. A human can dig around and figure out a bare error code. An agent gets a bare error code and it's stuck, or worse, it confidently does the wrong thing.

Non-interactive and deterministic. It runs without surprise prompts, so the agent never gets stuck waiting. Nothing kills an agent run like a tool that suddenly asks "are you sure? [y/N]" and sits there forever, because there's nobody to answer. Give it a flag to run straight through. Make it deterministic so the same input gives the same result.

Here's what I want you to notice about that list: every item on it is plain good CLI design. None of it is agent-specific magic. A human wants clear errors and predictable behavior and discoverable commands too. The agent just can't shrug and work around the absence of them. So building for the agent is building the tool well and refusing to lean on "eh, a human will sort it out."

Now the part people get wrong. They think designing for agents and designing for humans pull apart, that you're serving two masters and someone loses. They pull together. Here's the actual shape of it.

You design a really good core that works. Then you expose it. You add the flags: non-interactive, or `--json`, or introspect, whatever the tool needs. And now the agent can use the CLI tool, and the humans can use the CLI tool. Same tool. Same core. You built one thing and exposed it twice.

Then it gets extended even more from there. More UI for humans to use. Or just tooling with no UI. Or plugins that humans make, or plugins that help agents, different things like that. The extension is where the audiences actually diverge: a human wants affordances, a nice surface, a UI; an agent wants introspection and plugins and modes. Fine. Build those. But build them *on top of* the shared core, as extensions, not as two separate products you have to keep in sync forever.

At the core it's the same thing. That's the line I keep coming back to. It just has to be exposed to agents and to humans. So yes, there *is* an extra step. You don't get the dual-use property for free; you have to deliberately expose the core both ways.

That reframe is the whole point of this chapter, because it dissolves the objection before it starts. People resist building for both because they price it as double the work: two designs, two test suites, two things to maintain, two things to break. Really it's one core plus an exposure step. The core is the expensive part, and you were going to build it anyway. Exposing it for an agent is mostly the discipline of structured output and consistent commands and good errors, the things you should've done regardless.

There's an order question lurking here. Does the core come first, or do you design for both at once? It's enough of its own thing that it gets its own chapter later. For now: you expose the core both ways, and that exposing is the cheap part.

My own tooling is built exactly this way. Take fledge. It has a real `introspect` verb, and the entire job of `fledge introspect --json` is to let an agent discover the available commands as structured data instead of scraping `--help` text meant for a human. That's the exposure step made literal: same core, but the agent gets to ask "what can I do here?" and get a machine answer back. Then I make the commands themselves run headless. Set `FLEDGE_NON_INTERACTIVE` so nothing stops to ask a question, pass `--json` so the output comes back as data, and now the agent has a first-class way in that never once depended on reading prose.

Let me make it concrete. `fledge doctor` checks your project environment. Run it plain and you get something for your eyes: aligned columns, a checkmark per row, an emoji status, a summary line:

```
Git
  ✅ git 2.45.2
  ✅ repository: initialized
  ✅ remote: origin ➡️ git@github.com:CorvidLabs/fledge.git
  ✅ working tree: clean

8 checks passed, 0 issues found
```

That block is exactly the “before” an agent chokes on. It’s pretty, and it’s a screen. To know the remote is set, the agent has to find the right row, recognize a green checkmark, and trust that the spacing won’t move next release. The status it cares about is an emoji buried in a column. In practice: the agent scrapes the aligned text, the parse breaks when a longer repo name shifts the columns by two characters in the next release, and the agent reads the remote check as passing when it’s missing. It took the wrong branch on a string-match that was never a real contract.

Run the same command with `--json` and the same checks come back as data:

```
{
  "schema_version": 1,
  "action": "doctor",
  "passed": 8,
  "failed": 0,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "git", "status": "ok", "version": "2.45.2", "detail": null, "fix": null },
        { "name": "repository", "status": "ok", "version": null, "detail": "initialized", "fix"
      ]
    }
  ]
}
```

The `schema_version` is the first field on purpose. It’s the thing the agent reads before anything else, because it tells the agent which shape the rest of the document is in. The day the output format changes, that number changes with it, and an agent that pinned to a version knows to adapt instead of silently misreading new data as if it were old. It’s how the contract survives the tool getting revised.

Past that, same core, the same checks ran. The human gets checkmarks and a summary line. The agent gets a `status` field it can branch on instead of scraping a green checkmark out of a column. The `fix` field is `null` here because nothing’s wrong; that’s the contract: `fix` is `null` on a passing check.

Now run it where something *is* wrong. Say the repo has no remote configured. The plain version dims a row and the agent is back to guessing; the `--json` version turns that into a record it can act on:

```
{
  "schema_version": 1,
  "action": "doctor",
```

```
  "passed": 7,
  "failed": 1,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "repository", "status": "ok", "version": null, "detail": "initialized", "fix"
        { "name": "remote", "status": "missing", "version": null, "detail": "no remote configur
      ]
    }
  ]
}
```

That's the branch the agent actually acts on. `status` flips from `"ok"` to `"missing"`, `detail` says what's wrong in plain words, and `fix` is now a populated string, a literal command the agent can run to repair it. The agent doesn't have to know what a missing remote means or invent a recovery; the tool that found the problem also handed over the fix. The human gets a dimmed row; the agent gets a `status` to branch on and a `fix` to execute. One command, exposed twice.

The test for whether you got it right is simple. Hand the tool to a person with no agent. Does it work, is it good? Hand it to an agent with no person. Does it work, is it good? If the answer to both is yes, and you didn't build the thing twice to get there, you did it right.

The bridge

Two chapters back I flagged a bigger version of this argument, the one about percentages: how much of the world's code is written by humans versus agents, and what happens as that ratio moves. This is that.

Pick the thread back up. We've had a decade-plus of 100% human-written code, every line in every repo written by a person, and that means tons of legacy code, tons of tech debt, a decade of it running everything. That's the world as it actually is right now.

And now we're using AI on top of all of it.

That's the hard part, and people don't sit with it long enough. There's just so much legacy and human stuff already there. The AI shows up to *that*.

And if we keep going, if we lean further and further on AI to write things, it becomes a real question. At what point does it flip to 100% AI-written code? Does it ever? And what do you do with the decade of human code that's already there? Do you just leave the legacy code as legacy and move on? Rewrite it? Pretend it isn't there? Nobody actually has a good answer, and most people aren't even asking the question. They're just bolting an agent onto whatever they've got and hoping.

That whole transition, from all-human to whatever-comes-next, is the thing the right tools are for. The tools are the bridge across it.

Ideally, you set up tools that *both* humans and AIs can use. You set it up to write and build using spec-driven development, spec-sync, and then fledge for the commands. That's the bridge. If you start using these tools, it's easier to introduce an agent into existing code. And if you start a brand-new project, it's easier for humans to drive too. The same setup helps the agent walk into a decade-old mess and helps a person stay in charge of something brand new.

It's specs, not better prompts or smarter agents, because a spec is a shared source of truth for both sides. The human states intent. The agent builds to it. It's the same problem as the last chapter, one level up. An agent fumbling through a human tool guesses at the commands. An agent dropped into a codebase with no spec guesses at the *intent*. The spec is what it reads instead.

And it keeps the code honest to the spec. This is the part that makes the bridge trustable instead of just hopeful. spec-sync keeps the spec and the code in agreement, enforced in CI, so the agent can't quietly drift from what was agreed. That matters more than it sounds. The failure mode everyone's scared of with agents isn't the dramatic one; it's the quiet one, where the agent slowly builds something that isn't what you asked for and nobody notices until it's

deep in. CI checking code against spec is the thing that catches the drift while it's small. The contract gets read by both sides, and a machine checks that nobody broke it.

Here's what that machine actually checks. The spec is a markdown file, a `*.spec.md`, that writes down the contract: the purpose, the public API, the invariants, the error cases. spec-sync matches that against the real code, both directions. An export the spec never documented gets flagged. A spec promise the code doesn't have is an error. It checks declared schema against the actual migrations the same way. It's structural: does the public surface and the schema line up with what was written down. It's not generated tests and it's not a fuzzy diff against the prose, and that narrowness is why I trust it: it claims the contract and the code agree on the shape of things, nothing more. In CI it runs as a real gate. It exits non-zero when they've diverged, and posts the drift right onto the pull request.

That sets up the division of labor I actually care about. The human owns intent. The agent owns the grind. The human stays in control at the intent level, what this should do and why, while the agent does the implementation. Each side does the part it's actually good at, with the spec as the line between them.

And it's a safe on-ramp into legacy code, which is where this loops back to the percentage problem. You've got a decade of code with no specs, because nobody wrote specs, because a human wrote it and the intent lived in that human's head. The move is: capture what a piece *should* do as a spec, then let the agent work against that. You're not asking the agent to divine intent from ten-year-old code. You're writing down the intent first. A human can do that, it's the part humans are good at. And now the agent has a contract to build and refactor against. That's how an agent gets into existing code without it being a disaster. The spec is the on-ramp.

Picture the move on a real function that has no spec. Before: working code, no written contract, the intent living entirely in the original author's head, so an agent dropped into it reads the signatures, guesses at the invariants, and starts changing things. After: the same code with a `*.spec.md` sitting next to it, the public API and the invariants written down in plain language, and spec-sync enforcing in CI that the code stays honest to what's written. You do not have to do the whole codebase at once. You do it one piece at a time, writing the contract for the part you are about to hand over.

The spec didn't change what the code did. It changed what the agent could do with the code: work against a written contract instead of guessing at one.

fledge is the command surface for all of it: the consistent verbs both the human and the agent drive the work through, the thing from the last chapter that doesn't make the agent guess at how to build and test and run. spec-sync is the contract; fledge is how you act on it.

Here's the seam between them. spec-sync stands on its own: its own tool, its own CI check. But fledge treats the spec as a first-class part of the dev loop. There's a `spec` verb right alongside

the rest, and `fledge ask` and `fledge review` are spec-aware, so the spec is the context the agent reads when it answers a question or reviews a change. The spec isn't documentation the agent ignores; it's the thing fledge hands the agent to work from.

So the gate fires from both ends. fledge's `spec` pillar runs the spec-sync check right there in the dev loop, and the `CorvidLabs/spec-sync@v4` GitHub Action runs it again in CI, so nothing merges that quietly drifted. When an agent is doing the work, that check rides inside its loop, so it stays on-spec as it goes instead of finding out at the end.

Now follow it to the end, because the end is the reason for all of it. If this works, if the tools and specs and the rails that keep agents honest are just *there*, then humans move up. Up to intent and direction, the agent doing the grind underneath against a spec, in the open where you can check it. That move gets its own chapter later on; here it's enough to know it's where the bridge leads.

The percentage will keep moving. More of the code will be written by agents; that's just true. What's up to us is the bridge, the shared tools, the shared contract, and whether we actually build it. I don't think most people have started.

The identity wall

Before we get into the tools, the honest limit. This whole argument runs into a wall, and it's not one I can knock down with better tooling, so I want it said plainly and said early: the platforms won't let an agent exist as itself. Everything after this chapter is what you do given that.

If an agent is a first-class user of your tools, sooner or later it has to be a first-class citizen of the platforms too. Those are the same idea pointed in two directions. A tool that treats the agent as a real user gives it structured output and discoverable commands and a way in that doesn't depend on a human. A platform that treated the agent as a real participant would give it an account, an identity, a legitimate place to exist among everyone else's. That second thing is exactly what you can't get.

I tried. I went to give an agent its own identity on the platforms everyone builds on: its own account, not a wrapper around mine. The platforms don't allow it. I tell that story properly in the Building Agents book. The thing to take here is what kind of wall it is. The blocker in front of a genuinely autonomous agent is not the model's capability, and it's not even safety. It's that the platforms won't grant an agent an identity to exist with.

It's a policy wall, not a technical one. I can run the VM. I can wire up the loop. The model can do the work. None of that is the blocker. The blocker is that I can't make the platform allow the agent to be there, and that's not mine to fix. It's a decision someone else made, sitting upstream of everything I can build.

It matters because of accountability, which is the model I actually want. The end state isn't an agent running wild. It's an agent with a real, named, scoped identity: full capability, reduced privileges, a human approval gate. It does the work, it ships it as far as a pull request, and the merge is mine, because if something acts in the world under my name I'm the one who signs for it. But you cannot have *accountable* without *identity*. An agent you can't name, can't scope, can't point at and say "that one did this": there's nothing to hold accountable. Deny the identity and you've denied the accountable version along with it, and what's left is either no agent or an unaccountable one.

So here's the frame for the rest of the book. The agent can be a first-class user of a *tool* I build, because that part is mine. It can't be a first-class citizen of a *platform* I don't own, because that part isn't. Everything after this chapter is built inside that limit: this is what first-class for both looks like when you can give the agent a real way into your tool but you can't give it a real place in the world. Not the AI, not the tech, not safety. The platforms won't let the agent exist. Everything else I can build. That one I can't, yet. In 2026 the wall is still standing. The only ways through are workarounds you run yourself: an aged human account with real activity

history (a fresh agent account gets blocked immediately), or a verification layer you host yourself. That's the state of it.

When it’s bolted on

It’s easy to nod along to “build for both” and never actually picture the failure. So let me put you on the agent’s side of a human-first tool, because that’s where you feel it.

Two things go wrong, and they go wrong constantly.

The first is the agent gets stuck. Tools hang on interactive prompts all the time: a confirmation, an “are you sure?”, something sitting there waiting on a keystroke. And there’s nobody there to press the key. So the run doesn’t fail, exactly. It just stops. It sits at a prompt that was written for a person who’d glance over and hit enter, and the agent waits, because waiting is the only thing the tool gave it to do. A whole run dead on a question nobody’s there to answer.

The second is the docs. There either aren’t any, or there are and they’re confusing. Which means the agent has to *learn* the tool. And here’s what I keep noticing: the agent doesn’t really learn the tool. It can’t. There’s nowhere for that knowledge to live between runs. So it does the expensive version: it scans the files, reads whatever’s in the index, takes its own notes, reconstructs how the tool works from scratch. Every time. The tool *knows* what it can do, it’s right there in the code, and it just never tells the agent. So the agent rebuilds that picture from nothing on every single run.

Think about how wasteful that is. A person reads the docs once, maybe skims them, and carries it around in their head after that. They build up a feel for the tool. The agent gets none of that for free. Whatever the tool won’t tell it directly, it has to go dig up again, pay for again, guess at again. The work the tool should have done once, the agent redoes forever.

Both of these are the same mistake wearing two outfits. The tool assumed a human would be there, a human’s patience at the prompt, a human’s memory of how the thing works, and an agent has neither. It can’t shrug and wait. It can’t remember across the wall between runs unless you hand it something to remember.

And both of them map straight onto the list from a few chapters back. Non-interactive: don’t stop and wait for a person who isn’t coming. Discoverable: tell the agent what you can do, in a form it can read, so it doesn’t have to go reconstruct you from your own source code. The hang and the re-learning aren’t exotic agent problems. They’re just those two properties missing, and a human quietly covering the gap every time so you never noticed the gap was there.

Bolting agents onto human-first tools mostly works, right up until you watch what the agent has to do to make it work. Then you see how much of the tool was leaning on a person the whole time.

Here is one of mine, named. fledge began as a task runner I drove by hand. The core was clean from the start, the build and test and run logic sat behind the commands rather than inside them, so the day I first pointed an agent at it most of it just worked. Except one spot. The agent ran `fledge work commit` and stopped dead. That command prints `Commit message:` and waits for someone to type. There was no someone. The agent sat on a blinking cursor until it gave up, because that one command had quietly been leaning on a person the whole time. The fix was not a cleverer agent. It was the non-interactive path: a `FLEDGE_NON_INTERACTIVE` switch that makes every prompt behave as already answered, and a way to pass the message up front instead of waiting for it. The tell is that the prompt only ever existed because I had been the one answering it. Built for both from the first commit, it would not have been there to hang on.

Expose the core twice

Here's the heuristic. If the core is hard and novel, build it first and expose it after. If the core is straightforward, design for both from the first commit. Here's why it works that way.

When the core is the hard part, the novel part, the thing that's genuinely difficult to get right, I build that first and worry about the surface later. Crypto is like this. A parser is like this. A scorer that has to be right is like this. The hard middle is where all the risk lives, so that's where the attention goes first. I get the thing *working*, correctly, for one real case, before I think much about how an agent or a human reaches in to use it.

And the reason that order is fine, the reason it's not a gamble, is that once the hard core is right, the surface is cheap. Adding `--json`. Adding an introspect command. Adding a non-interactive flag. None of that is the hard part. It's a few hours of exposing what's already there in a shape the other side can read. So building it later doesn't cost me much. The expensive thing got built first, the cheap things got bolted on after, and that's the right way around.

But plenty of tools aren't like that. Plenty of the time I already know both audiences are coming, because I always know both audiences are coming now, and so the agent-facing shape pulls on the design from the first commit. I'm not building a core and then discovering an agent needs it. I'm building it knowing a human will drive it by hand and an agent will drive it headless, and both of those are in my head while I'm shaping the thing. There's no "expose it later" step because exposing it was never a separate phase.

So the real picture is: the expensive part gets built first, and the expensive part is usually the core. When the core is hard, you nail it and the surface follows easily. When the core is straightforward, there's nothing to protect by going in order, so you just design for both at once. Either way you land in the same place: one good core, reachable cleanly by a person and by an agent.

The easy objection to that claim is a CLI tool where the human and agent surfaces are nearly identical anyway. Fair. The harder objection is a tool where the human surface is inherently visual, stateful, or interactive: a design tool with a canvas, a REPL, an interactive debugger. Something like Figma. The human surface of Figma is a canvas you drag things around on. The agent surface is a REST API and a plugin interface. They look nothing alike. If the principle "one core, two surfaces" holds anywhere, it has to hold there too.

It does, and the reason is that the REST API Figma exposes to agents is the same structured layer the canvas is built on. When you drag a frame, the canvas calls the same document model the API reads and writes. The human surface is rich because that core is rich. The agent surface

isn't a second system bolted on for agents; it's that same foundation with the canvas taken off the front. The gap between the two surfaces is real. There's still one core underneath both.

Now the claim the whole book leans on, the one I owe you honestly: building for both is not double the work. Here is the actual reasoning, not a number I made up. Double the work would be two cores: two implementations of the hard part, two test suites over the real logic, two things that can be wrong in different ways. That is not what this is. There is one core. The hard part exists once. What you add for the agent is exposure: structured output, a non-interactive flag, a way to introspect the commands. That exposure is cheap relative to the core, and it is cheap for a reason you can check: it has no new logic to get correct. `--json` serializes a value the core already computed. A non-interactive flag skips a prompt the core never needed. Introspect reports commands that already exist. None of it re-derives the answer; it re-presents an answer you already have. The expensive thing about software is being correct about the hard problem, and you pay that once.

There's a real cost to be honest about, and it lands in two places. The first is when the core was already built human-first. Then exposing it for an agent isn't free: you have to go find every place the logic leaked into the presentation (a value that only ever existed as a formatted string, a decision made inside a print statement) and pull it back into the core so there's something real to serialize. That refactoring is the retrofit cost, and it's exactly the cost this whole book is arguing you can avoid by building for both up front. The second is the testing surface: two ways in means you do want to check both ways work, and that is more test cases than one way in. But it is more cases over the *same* core, not a second core to verify. The logic you're testing is shared; you're confirming two surfaces expose it faithfully, which is cheaper than proving two separate things correct.

fledge is the case I can cost out, because it is the one I built before I was thinking about agents at all and then watched an agent meet it. The core was already clean, the logic lived behind the commands, so exposing it for an agent was a thin layer and not a second build: structured `--json` output, a headless mode, an introspect command that lists what is available. That work took days, not weeks, and it took days for the reason this chapter has been arguing. There was no second core to write, only a second door onto the one already standing. What was harder, the only thing that was harder, was the handful of commands where I had let a decision live inside a prompt instead of inside the core. Those I had to pull apart so the choice sat somewhere an agent could reach. That was the whole bill, and it was small, and it was exactly the retrofit tax named above, paid down to almost nothing because the core had been kept honest first. What was cheaper was everything else, which is to say nearly all of it.

What I'd warn you off is the inverse: starting from "how do I make this agent-friendly" before the core is any good. That gets you a thin tool with a JSON flag taped to the side, which is just the human-first-with-agents-bolted-on problem again. The surface is cheap *because* the core is

solid. If you skip the core, the cheap surface has nothing under it, and you find that out the first time anyone leans on the tool for real.

Why I build my own stack

Fair question to ask me at this point: why build any of it? fledge, spec-sync, corvid-ai, there's existing stuff. Why not just wire together what's already out there and get on with it.

A few reasons, and they're all true at once.

The first is that the domain is brand new. Building tools *for an agent-and-human world* is not a solved thing you can go shopping for. Almost everything out there is human-first, or the newer stuff is agent-first, and the thing I actually want, first-class for both, from the start, mostly doesn't exist yet. So I'm not reinventing wheels. There aren't wheels. If I want a tool built on the assumption I keep going on about, I have to build it, because the people who came before me were building for a different world.

The second is that building on my own stack proves it. This is the part I care about more than it might sound. You can write a beautiful README claiming your tooling is good. Nobody should believe you. What they should believe is that you built real things on it and the things work. So I do. I build on fledge and spec-sync and corvid-ai because carrying real weight is the only honest test of whether they hold up. If the stack is good, the things I build on it are good, and if the stack is bad, I find out fast, because I'm the one stuck with it.

The third is plain: building it is how I understand it. I trust a dependency only if I could have written it myself, and that's not a slogan. It's a decade of receipts. AppState, the state-and-dependency-injection library I still maintain at 3.0; Cache; Fork for parallelizing async work; the single-letter `c` / `o` / `t` composition primitives under 0xOpenBytes; CacheStore, the SwiftUI thing that turned into AppState. Years of small Swift libraries, each one a thing I built because I wanted to understand it to the bottom, not point at a black box and hope. Building the thing is how the knowledge gets into my hands instead of staying as a vague idea of what some library probably does. The tools in the stack are tools I can open up and change, because I put every part of them there.

There's a fourth reason, and it's narrower than it sounds: I want to be early to this. The space is new, the wheels don't exist yet, and I'd rather build the tools for it and be wrong about some of them than wait for someone to hand me the right ones. That's the bet.

None of these would carry it alone, but together they're why I'm not just gluing other people's tools into a pile. The tools are the point. They're the thing I'm actually trying to be good at.

Two kinds of certainty

There are two different things people lump together as “how sure are we about this code,” and I want to pull them apart, because I care about both and they’re not the same thing at all.

The first is risk, and risk I want static. Deterministic. No model in the loop. This is what augur is for. You hand it a change and it scores how risky that change is, the same way every time, on signals it can name. And “signals it can name” is the whole point, so let me name them: does the diff touch sensitive ground (auth, crypto, payments, migrations, CI, dependencies), did code change without tests changing with it, are these churn-prone files with a history of reverts, does anyone actually own them. Each one is a yes-or-no you could check by hand. Add them up with documented weights and you get a number, not a vibe. The reason it has to be static is the whole reason it’s worth anything: if the thing deciding whether code is dangerous is itself a language model giving you a feel for it, you haven’t measured the risk, you’ve just moved the guessing one box over. A risk score you can trust is one that says the same thing tomorrow that it said today. So that one stays a fixed, repeatable thing you can reason about. I get into how it actually works in the Building Agents book; here the point is just that risk is the *static* side, and it’s static because it’s a sum of named signals you can point at.

The second is confidence, and confidence I actually want *from the agent*. This is the one I love, and it’s the opposite of static. It’s the agent telling me how sure it is about the thing it just did. And the reason I love it isn’t really the number. It’s what asking for the number does to the agent. When you make an agent put a confidence rating on its own work, it has to stop and look back at what it did. The rating reframes the work for it. It can’t just produce and move on; it has to turn around and assess.

And the part that makes it genuinely useful is the granularity. An agent will happily give you one confidence number for the whole change. Fine, but that’s almost too coarse to act on. Where it gets good is when you narrow it down. A confidence rating on every file. On every individual change. Now you’ve got the agent’s own read on exactly which parts it’s sure about and which parts it isn’t, and that’s the map you actually want. It points you right at the spots the agent itself is nervous about, in its own words, before anyone else has looked.

So these are two different instruments. Risk is static, deterministic, mine to trust because it never moves. Confidence is the agent’s, alive, useful precisely because it comes from the thing that did the work and made it look again. The mistake is to blur them: to call the static risk score “confidence,” or to expect the agent’s confidence to be deterministic. They’re answering different questions. One asks “how dangerous is this change,” and you want a machine that can’t be talked out of its answer. The other asks “how sure are you about what you just wrote,”

and you specifically *want* the one who wrote it to answer, file by file, because the asking is half the value.

The only time having both actually matters is when they disagree. A change that scores low risk but where the agent rates its own confidence low is exactly the signal you'd miss with one instrument. Low risk means the signals were quiet: no auth, no migrations, no churn. But the agent's low confidence means it knew it was guessing at the logic, that something in the change felt uncertain even though nothing it touched was categorically dangerous. That is the change one instrument waves through and the other stops on, and it is the whole reason you keep both.

Keep them separate and they both work. Mix them up and you get a deterministic gate you can't trust because a model's in it, or a reflection step you've drained the life out of by demanding it never change. So I build them as two separate instruments. The risk gate stays fixed; the agent's confidence stays alive. That's the only way to get both.

The contract is the spec

The spec is the thing that sits between the human and the agent, and I've already said why it matters. It's the shared source of truth, the thing that stops the agent drifting, the on-ramp into legacy code, the line that keeps the human in charge. All of that holds. What I want to do here is go one level down, into what a spec actually *is* when you build this way, because there's a shape to it that's easy to miss.

Before the theory, here's what one actually looks like. A `*.spec.md` for spec-sync is a markdown file with named sections, and a tiny one, say the contract for a single function that clamps a number into a range, reads about like this:

```
# clamp

## Purpose
Constrain a value to an inclusive [min, max] range.

## Public API
`func clamp(_ value: Int, min: Int, max: Int) -> Int`

## Invariants
- Result is always >= min and <= max.

## Behavioral Examples
- clamp(5, min: 0, max: 10) == 5
- clamp(12, min: 0, max: 10) == 10

## Error Cases
- min > max is a programmer error (precondition failure).
```

That's it: a few labeled sections, no prose narration. Real specs add the rest of the required sections (Dependencies, a Change Log), but the shape is already visible here: it states *what's true*, in a form a machine can hold the code against. Now the theory.

Start with what goes in the spec itself. The spec is the contract. Purpose, the public surface, the invariants, the behavioral examples, the error cases: the checkable shape of the thing. What it is, not a story about it. The second a spec turns into a wall of prose describing the code, it's dead, because prose drifts from code the moment either one moves and now you've got two things that disagree and no way to tell which is lying. So the spec is kept tight and contractual on purpose. It's the part a machine can hold the code against.

It's also intent, not implementation. The spec says what the thing should do and why it should do it. It does not say how. The moment a spec starts dictating implementation, it stops guiding

the agent and starts fighting it. You've taken the part the agent is good at, the grind of figuring out *how*, and you've pinned it down from above for no reason. Keep the spec at the level of intent and the agent has room to actually work. State what should be true. Let it build to that.

There's one more thing the spec has to do, and it's the part people skip: it has to say how you'd know. Intent isn't just what should be true, it's what you'd accept as proof it's true, and what would make you say it isn't. A spec that names the first and not the second is a spec the agent can satisfy in a way you didn't mean, building something that matches the words and misses the point, with nothing to catch it because you never wrote down what catching it would look like.

That's what the behavioral examples and the error cases are really for. `clamp(12, min: 0, max: 10) == 10` is an acceptance signal: run it and you know. The precondition failure on `min > max` is a rejection signal: it says what going wrong looks like, concretely, instead of "handle bad input." Both are observable, and neither needs me in the room to judge. So when you write the contract, write both halves. The acceptance signal is you deciding what done means ahead of time. The rejection signal is you deciding what broken means before you're staring at it and tempted to call it fine.

But here's the shape I think people miss: it's not one file. There's the spec, and then there are companion files around it, and each one carries a different kind of knowledge that the spec itself shouldn't.

Start with the spec itself. The spec is the one tied tightly to the code: the non-code picture of what the code actually does, close enough to it that spec-sync can hold the two together one to one. That's the required file, the checkable one. Around it sit companion files, and each carries a kind of knowledge the spec shouldn't.

I think of them by the kind of knowledge they hold rather than by any fixed filename. There's the **requirements** kind: the high-level one, written the way a product owner writes, the user stories, the "as a user, I want…", the business-level intent. There's a **context** kind: context for the agent, the stuff you just need written down somewhere so it has it. There's **design**: the design notes, the thinking, the why-it's-shaped-this-way that doesn't belong in the tight contract but you don't want to lose. And there's **testing**: how you'd actually verify the thing does what the spec says. Don't read those as four blessed file names; read them as four kinds of knowledge that want to live next to the spec instead of inside it. What matters is the split, not the labels.

And it runs both directions. A human can write the requirements and let the agent turn them into the spec; or a human writes the spec and the requirements fall out of it. Intent and contract, either order, with the agent able to move between the two.

The reason that split matters is that it keeps the contract clean while still giving the agent everything else it needs. The spec stays small and checkable, the part spec-sync can actually hold the code to. Everything that's real but *not* checkable (the business requirements, the design reasoning, the running context, how you'd test it) lives next to the spec instead of bloating it. So you get a contract that's tight enough to enforce, surrounded by the looser knowledge an agent needs to do the work well, and the two don't contaminate each other.

So the spec works as an interface, not as a document the agent skims and ignores. A tight contract it builds against, plus the companions that carry the rest, with a clean line between the part a machine checks and the part a human wrote down so nothing got lost.

Code is cheap, trust is scarce

Agents made code cheap. That's the fact that reorganizes everything else, so start there.

When a human had to type every line, writing the code was slow, and the slowness was doing a hidden job. You couldn't write a thing without partly understanding it. The act of writing was also the act of vetting. They came bundled, for free, because the same person did both at the same speed. That bundle is what just broke. An agent hands you a forty-file pull request before your coffee's done, and the writing and the vetting come apart. The code got produced. Nobody understood it on the way out. Producing it stopped meaning anyone vetted it.

So the hard part swaps. It used to be the code itself, every line written by hand. Now code is cheap, there's as much of it as you want, and the scarce thing is trust. Who actually looked at this change, and how hard, and should it be allowed to land. That's the question that's expensive now, and it's the question the tooling has to answer, because the old answer ("well, somebody wrote it, so somebody understood it") isn't true anymore.

Which means review has to change shape. You cannot rubber-stamp a forty-file PR, and you also can't honestly read all of it, and pretending you did is the actual danger. So the tools have to triage your attention: point you at the risky part and let you spend your judgment there instead of spreading it so thin across the whole thing that it's worth nothing. The scarce resource is a human paying real attention, and you spend it where it counts.

This isn't hypothetical for me. `fledge review`, which runs the change past a model through my corvid-ai client, caught a real bug in my own work that would otherwise have shipped, and spec-sync has caught real drift between code and its spec more than once. That's the receipt I trust more than any argument: the trust layer found things a faster me would have waved through. The everyday version of it is a lane I call `verify` (format, lint, test, build) that runs before anything merges, the same gate the agent's own runner has to clear. None of that speeds up the writing. All of it is the work the cheap writing shoved downstream, made into something that actually runs.

And once agents are landing changes, you need a record. A portable trace of who or what vetted each change, and it can't live in some SaaS dashboard that gets switched off, because then your provenance disappears the day someone stops paying the bill. It has to ride along with the code itself. When an agent lands something, there should be a durable answer to "who vouched for this," and that answer has to outlive whatever tool produced it.

But there's a piece bigger than trust, and it's the one people skip. Testing. Actual testing, setup, the whole apparatus around the code. Here's the trap: agents make you ten times faster at *writing*, and nothing else automatically speeds up to match. The tests still have to be written

and run. The setup still has to happen. The review still has to happen. So if you let development go ten times faster and leave everything else at its old pace, all you've done is move the bottleneck. The code isn't the slow part anymore; the testing is, the verification is, the trust is. You can't ten-x the writing and not ten-x everything around it. The work didn't go away. It moved downstream, onto the parts that were never the bottleneck before and suddenly are.

Cheap code doesn't make the work disappear. It makes the writing cheap and shoves the weight onto everything that decides whether the cheap writing is any good. That work was always there. It was hidden behind how slow writing used to be, and now the slowness is gone and there it is: the whole job, standing where the easy part used to be.

Humans move up

If the tools and the specs and the trust rails are just *there*, ordinary, the default way things are built, then the human moves up. Up to intent. You stop being the person who types the implementation and become the person who decides what should exist and why. Writing the code by hand becomes optional instead of being the job. That's the direction all of this points, and I want to be careful about how I describe it, because it's easy to round off to the scary version.

The headline is not "agents autonomously run everything." The human moves up a level and the agent does the grind underneath, against a spec, in the open where you can check it. The human gets elevated; nobody gets replaced. What you get there is a real team, handing work back and forth, each side doing what it's actually good at. And it becomes the default. Not my stack, not a niche a few people do. Just how building works.

Here's the thing I keep landing on about why the human stays in it. Most of the good things AI generates right now have a person in the loop making them good: choosing, correcting, deciding what counts as good in the first place. And yeah, at some point the models get good enough to make good things more or less on their own. But there's a core thing humans have that I don't think falls out so easily: we're good at driving. At intent and purpose. An AI doesn't really have a purpose of its own, not until it's given one, not until it finds one. It needs a human to *be* the purpose. The model can do the work once there's a why; it doesn't generate the why. That's the part that stays ours longer than the typing does.

Now let the picture run forward, because it goes somewhere strange and I think it actually goes there. You run your own agents to do the work. An agent can live on its own and just do stuff. Maybe you point it at a general area to look into, maybe it talks to other agents, maybe you put some money into it and it goes off and works, talks to other agents, pays other agents for what they do. People run agents and the agents make money. That's not a side feature. That's agent-driven development as an actual economy, with humans setting the purpose and the agents grinding away underneath, trading with each other.

And in that world the humans are the ones making sure it all works and that someone still understands it. Because at some point the code itself stops mattering in the way it does now. You're not reading every line, the agent wrote most of it, the volume's past what any person tracks. Fine. But here's the line I won't give up: a human still has to be able to get into the code and change it. Has to. The day you can't open it up and fix it yourself is the day you've handed something away you shouldn't have.

Which is why the code has to stay clean. Clean at every level, readable and changeable by a human and by an agent, all the way down. First-class for both was never only about today's

brittle little agents fumbling today's tools. It's the thing that has to hold even in the version where agents do most of the building. *Especially* there. The only way "the code won't matter" doesn't quietly become "you've lost control of the code" is if the code stayed clean enough, the whole way down, that a human can still open it up and change it. That's the whole job, kept open.

Start Monday

Say you've read all this and you buy it. Humans and agents on the same tools, first-class for both, the spec as the contract, the trust rails. What do you actually do Monday morning? Here's the honest answer, in the order I'd do it.

Point an agent at one of your tools and watch it choke. That's the first move, and it's the one that teaches you the most, because the agent is the discipline. You think your tool is fine. You've used it for months, your hands know it. Hand it to an agent with no hands and no eyes and no memory between runs, and every gap you'd been quietly covering for is suddenly visible. The prompt it hangs on. The output it can't parse. The command it can't discover. You don't have to guess where your tool assumed a human. The agent shows you, immediately, by failing exactly there. Fix what you find. That single exercise will teach you more about building for both than this whole book did.

Then go use the tools that are already built this way, so you're not starting from a blank page. spec-sync, for the contract. fledge, for the commands. augur, for the risk. attest, for the record of who vouched. They're not the only way to do any of this, but they exist, they're built on the assumption this book is about, and they'll show you the shape of it working instead of you having to derive it. Use them on a real project. Feel where they help and where they don't.

And bring your agent into that. Have it actually learn the tools and use them: let it drive spec-sync and fledge, let it work against a spec, let it run the checks. That's the loop you're trying to build: a human setting intent, an agent doing the work through tools that treat it as a real user, the spec keeping it honest. You don't get that loop by reading about it. You get it by running it on something you care about and watching where it holds.

What you're really doing, under all of it, is building and collaborating on tools and trust. That's the whole game. The tools, so a human and an agent can both do the work first-class. The trust, so you can actually believe what landed. Those two things are what this entire book has been circling, and they're the two things worth your time to build.

Point an agent at one of yours. Watch where it chokes. That's where you start.

About the Author

0xLeif (leif.algo) builds in the open. A decade of small, composable Swift libraries like AppState, Cache, and Fork. The CorvidLabs lab. A stack of agent tools that mostly started as “I wished this existed.” Off-keyboard he is Zach Eriksen.

These books are interviews, shaped into chapters and checked against the real code.

github.com/0xLeif · leif.algo

Acknowledgments

Thanks to CorvidLabs, for being the room where these ideas get tested and argued into shape.

Thanks to the open-source maintainers whose tools this whole stack stands on. None of this gets built alone.

And thanks to the early readers and the pay-what-you-want supporters who make “free online” something I can keep doing.

Colophon

Set from Markdown, built with bookgen, a small pure-Rust pipeline (no Python).

Interview-driven and AI-assisted; edited and fact-checked by hand. Written without em dashes. Cover and chapter art from the Corvid and Nature collections on Algorand.