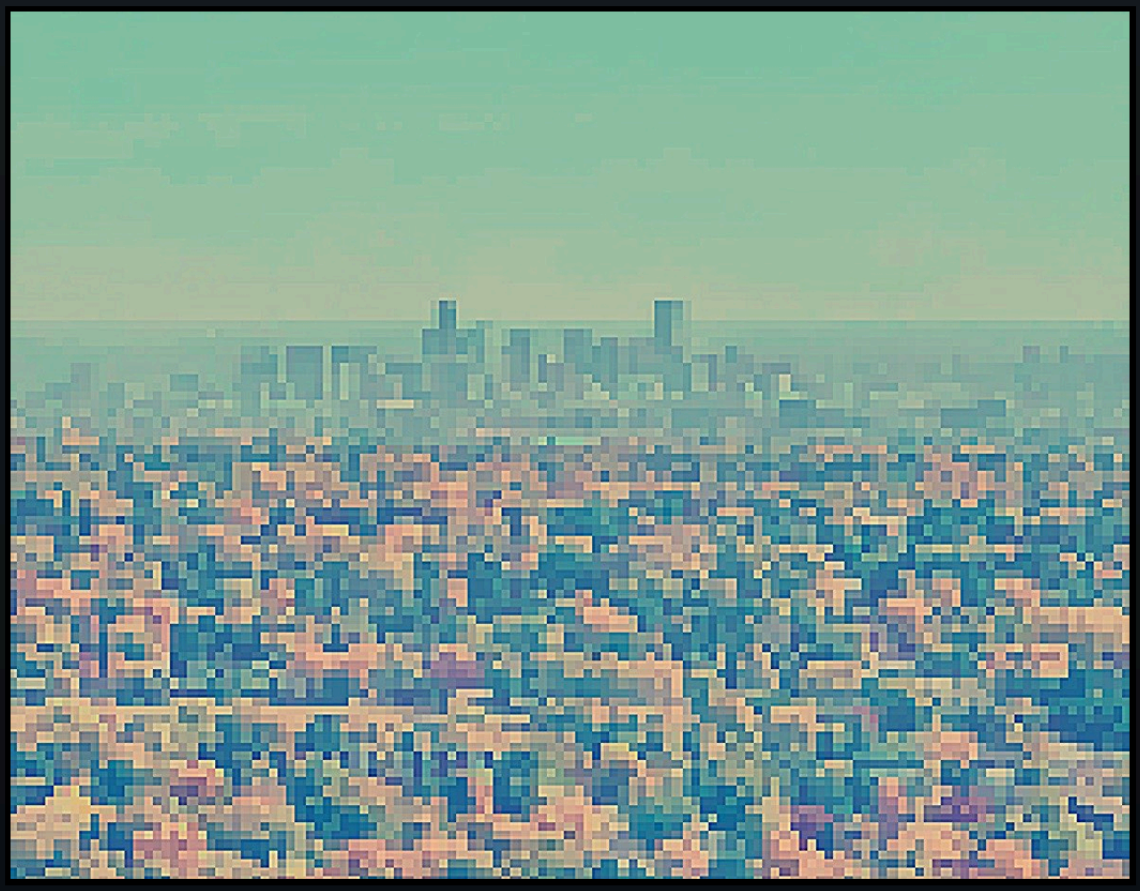

Open Source Tooling

Building tools people actually use

ZACH "LEIF" ERIKSEN

Open Source Tooling

Building tools people actually use

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
- **Building Agents**: Notes from trying to give software its own hands
- **Open Source Tooling**: Building tools people actually use *(this book)*

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

Contents

Introduction

This book is about the craft of building tools that other people, and other agents, actually use.

It is built around fledge, one command-line tool that covers the whole development lifecycle, and the question it kept forcing me to answer: what makes a tool worth depending on? The claim is that the answer is the same for a human and for an agent. A good tool has one clean surface, no hidden interactive steps, a real plugin boundary, and a spec that stays honest as the code changes. Build that, and an agent can drive it on day one, because it was never built to need a person in the first place.

This is the second of the two evidence books. *First-Class* argues that software should be first-class for both. The *Field Guide* turns that into a method. *Building Agents* shows the agents. This shows the tooling underneath them, down to why Rust was the right reach and how the WASM sandbox keeps a plugin from doing something it should not.

It is for anyone who has ever shipped a tool and watched it get used in ways they did not plan for. You do not need fledge. You need the habits that make a tool survive contact with users who are not you, including the ones that are not even human.

One CLI for the whole lifecycle

Every repo had a different dialect. Different Makefiles, different scripts, different READMEs, each one with its own idea of how you build, test, and run the thing. None of it transferred. You'd open a project you hadn't touched in a while and the first job was figuring out the local incantation all over again. Which script, which target, which order. The work itself wasn't hard. Running tests isn't hard. The problem was that it was different everywhere, and the difference was pure overhead.

So I wanted one consistent interface across all of them. That's where fledge starts. The line on the repo says it plainly: *one CLI, your whole dev lifecycle.*

There were really three things pushing me toward it.

The first was that sameness problem, every repo speaking its own language. The second was bootstrapping. I kept copy-pasting the same setup and scaffolding into every new project. Same pile of files, same wiring, over and over, before I could even start on the actual thing I wanted to build. The third was scale. I'm spinning up tons of projects, with agents working on them, and I needed a single CLI they could all rely on. Not "a tool I use." A tool that everything and everyone working across all those repos could lean on the same way.

That last one is easy to miss. fledge isn't born from one repo that annoyed me. It's born from having a lot of repos, a lot of projects in flight, and agents in the loop on them, and needing all of that to talk to one consistent surface instead of a hundred bespoke ones. The agents-can-rely-on-one-CLI angle is its own thread, and I get into the agent side of it in the agents book. Consistency, scaffolding, and scale: that's the origin.

Built for humans and agents both

There's an idea under all of this, and it's bigger than fledge. A lot of my tooling comes from believing humans and agents are going to use the same tools.

Right now most of what exists is human-first. Projects get built for people, and then we try to bring agents into them after the fact. There can be agent-first stuff and there can be human-first stuff, but what we actually need is to make agent-and-human-first projects, built from the start so both are first-class.

That's what all my tools are. A tool should work first-class either way: a human can use it without an agent, and an agent can use it without a human. Neither one is the afterthought. And when an agent does use it, the tool should *help* it, not leave it guessing at all the commands and how everything works.

Designing for humans and agents as equal first-class callers is the real reason fledge looks the way it does, and it's the thing to keep in mind for the rest of this book.

What zero-config actually means

When I say zero-config, I mean you don't have to teach the tool about your project before it can help you. You drop it into a repo and it works. There are three pieces to that.

It auto-detects the project and its commands. Swift, Rust, Node, whatever. It figures out what kind of project it's looking at and knows the right build/test/run for it. No setup step where you describe your project to a config file first.

The same verbs work across every project. `build`, `test`, `run`, `lint`, the same words, regardless of what's underneath. That's the direct payoff of the "every repo had a different dialect" problem. The dialects are still down there; Cargo is still Cargo and Swift Package Manager is still Swift Package Manager. But you stop having to care which one you're standing in front of. You learn the verbs once and they're the verbs everywhere. The detection handles the translation down to whatever the real underlying tool is.

This is the humans-and-agents idea made concrete. A person stops having to relearn each repo. An agent stops having to *guess*. It doesn't have to go hunting for the right command for this particular project, or read the README and hope. It runs `build`, it runs `test`, and the tool already knows what that means here. The thing that saves me the relearning is the same thing that keeps the agent from guessing. fledge was built first for me and the agents I run, and it earns its keep there before anywhere else.

And it's extended via plugins. A fresh repo gets the core, and you add capabilities by dropping in plugins. The core knows the common verbs and how to detect the common project types; everything past that comes from plugins. So zero-config doesn't mean "does everything out of the box." It means the part you get out of the box needs no configuring, and you grow it from there by adding plugins, not by writing config.

Concretely, a first run goes like this. You drop into a repo and run `fledge`, and it auto-detects the project (Swift, Rust, Node, whatever) and just knows the right build/test/run for it, no config step. Ask it to introspect and it *tells you the available verbs* for this repo: it's self-describing, so you (or an agent) don't have to guess what's possible. If it's a brand-new project, the first real move is usually scaffolding from a template instead of detecting an existing one. And the whole thing works headless from the first command: set `FLEDGE_NON_INTERACTIVE`, ask for `--json`, and an agent drives it from step one. Detect, introspect, maybe scaffold, then run.

The lifecycle part

The repo calls fledge a zero-config task runner with scaffolding and AI review. The task running is the verb set above: the day-to-day build/test/run/lint. The scaffolding is the answer to the second origin problem: instead of copy-pasting the same setup into every new project by hand, fledge stands it up. And there's an AI review piece baked into the lifecycle too.

Task running came first. It's the seed, the part everything else grew onto. Scaffolding is really templates: stand a new project up from one instead of copy-pasting. And AI review got folded in because review is part of the dev loop. If agents are writing the code, grading it isn't a separate ritual you go run somewhere else, it's just another verb next to build and test. One surface beats three tools, for me and even more for an agent that would otherwise have to learn three things. These three are the structural pillars, the core, with plugins as the extension layer around them.

fledge's own one-liner is "one CLI, your whole dev lifecycle, zero-config task runner, project scaffolding, AI review, and more," and that's exactly the three pieces. It's a single Rust binary, and you get it the boring obvious way: `brew install CorvidLabs/tap/fledge` off the Homebrew tap, or `cargo install fledge` if you'd rather, or a shell script. Nothing exotic to install it.

How I actually use it

This isn't a tool I reach for sometimes. It's every repo, all the time, the default way I build, test, and run everything now. And my agents drive it. The agents run fledge more than I do by hand. It got built to tame a lot of repos, and now it's the one CLI both I and the agents I've got working on those repos rely on, everywhere.

That's also why the next piece matters. A CLI that everyone leans on, that anyone can extend, that agents install and run, that can't be a fixed set of features I bless one at a time. It has to be extensible by anyone, in whatever language they want, without touching the core. And once you let arbitrary plugins in, and once agents are the ones running them, you have to think hard about trust. That's the next chapter.

Why fledge and not Make

People ask this the second they hear what fledge does. You built a task runner? There's Make. There's Just. There's Turborepo. There's the npm scripts block sitting right there in every package.json. Why write another one?

The honest answer is that the wall I hit was never Make. I came up as an iOS developer, and to automate a build there the road kept leading to the same place: fastlane, which means Ruby. I didn't want Ruby. I'm a Swift person. I wanted to automate my builds in the language I work in, and instead the whole ecosystem funneled me into a Ruby DSL and a `Gemfile` and a tool that was its own little world to learn. Every time I wanted to ship something the answer was "write a fastlane lane," and every fastlane lane was Ruby I didn't want to write, sitting in a runtime I didn't want to manage, doing a job I felt like I should be able to do in Swift. That's the pain fledge actually grew out of. Not "Make is clunky." It was "why am I forced into someone else's language to automate my own project?"

So when people line fledge up against Make and Just, they're aiming at the wrong target. The answer isn't that those tools are bad. What I wanted out of a task runner was three things, and the fastlane wall is where all three came from.

I wanted it my way, in my language, on my terms, not locked to one runtime the way fastlane locks you to Ruby. A task runner is the thing you touch a hundred times a day, and after a while you stop wanting to live inside someone else's choices about how it should feel, or which language you have to think in to use it. I wanted `build` and `test` and `run` to mean the same thing in every repo, and I wanted the steps underneath to be writable in whatever language the job called for instead of being marched into Ruby. Make doesn't stop you, but Make doesn't give it to you either. You build the consistency yourself, in every Makefile, by hand, forever. A tool that lets me reinvent the dialect per project hasn't solved my problem; it's just given me a nicer place to keep reinventing it.

The other two are the ones from the first chapter, and the fastlane file is where I'd have wanted them too. None of those tools is agent-first (JSON by default, introspection, a headless mode) which is the part I care about most now that my agents drive this thing more than I do by hand. And none of them is a single light binary that drops into a repo with no runtime to install first, which is exactly what a consistent surface across a pile of different-language repos has to be. A fastlane lane needs Ruby; npm scripts need npm; a runner I'd write in TypeScript needs a TypeScript runtime. fledge needs nothing.

None of which is "Make is bad." Make is great at what Make does, Just is a clean command runner, Turborepo caches a JS monorepo well. If one of those is all you need, use it. I'm not going to pretend fledge wins a feature-by-feature fight on their home turf.

But none of them was the thing I'd actually wanted standing in front of a fastlane file years earlier: one surface, agent-first, a single light binary, and the freedom to write the steps in whatever language fit instead of being funneled into Ruby. fledge is the tool I wished I'd had back then, finally built.

fledge and MCP

MCP is the ambient standard now. In 2026, agents mostly consume tools through MCP servers. If you build a tool and want agents to be able to use it reliably, the path of least resistance is to expose an MCP server. That's just the state of the ecosystem.

fledge doesn't have an MCP server. And yet agents drive it constantly across dozens of repos, and it works. The reason is worth being explicit about, because it tells you something about which piece to build first.

The CLI is the primitive

An MCP server is a production interface. It handles request routing, it logs, it gives you structured observability over what an agent asked for and what it got back. Those are good things to have. But an MCP server is a layer you put on top of something. The question is: on top of what?

If you build the MCP server first and the CLI is an afterthought, you get a tool that works for agents and is a pain for humans. If you build the CLI first, you get a tool that works for agents right now with no protocol adapter, works for humans, and can have an MCP server put in front of it whenever you need one. The CLI is the primitive. Everything else sits on it.

fledge was built with that in mind. Every command emits `--json`. There is `FLEDGE_NON_INTERACTIVE` for headless execution. `fledge introspect` gives any caller, human or agent, a structured manifest of every available verb, what it does, and what it accepts. There are no interactive prompts that block unattended runs. The tool describes itself.

That profile, structured output by default, an explicit capability manifest, no hidden interactive steps, is exactly what an MCP tool surface gives an agent. fledge has all of it without the protocol, because those properties came from designing for agent callers from the start, not from wrapping the tool later to make it agent-friendly.

A Claude instance driving fledge today calls `fledge introspect`, gets back a JSON manifest of what's available, picks the right verb, passes `--json`, and reads structured output. That is the same interaction loop an MCP server would mediate, minus the MCP framing. The CLI is already the clean tool surface.

MCP is not the competitor

The framing that sometimes comes up, "CLI or MCP?" treats them as alternatives. They are not. MCP is a transport and protocol spec. The CLI is the thing that does the work. The

question is not which one to have; it is which one to build first and which one to layer on top.

Build the CLI first, the kind that describes itself, emits JSON, and runs headless. Once you have that, exposing an MCP server on top is straightforward: you map each `fledge introspect` entry to an MCP tool definition, you call the CLI under the hood, you pass the JSON output back. The core is already structured JSON over a clean boundary with an explicit capability manifest. An MCP server is a production API on top of that same primitive, the same way a REST API might sit in front of a well-structured library. The interface changes; the work does not.

And because the CLI is the real surface, everything that can call a subprocess can use it without the protocol adapter. A shell script can use it. A CI runner can use it. A human at a terminal can use it. An MCP client can use it through the server layer. None of those callers requires the others to exist. You get universality from the primitive, not from the protocol.

What MCP adds

None of this is an argument against MCP. It adds real things once it's sitting in front of the CLI.

Logging and observability. An MCP server sits between the agent and the tool, so you can record every call, time it, inspect arguments, flag anomalies. The CLI invoked directly gives you whatever you pipe into a log file. The MCP layer gives you structured observability without instrumenting every command individually. For production use where you want to audit what an agent asked for, that matters.

Discoverability at the protocol level. MCP has a standardized way for an agent to ask "what tools are here?" and get back a structured answer. fledge has `fledge introspect` for the same thing, but `fledge introspect` requires knowing fledge is there. An MCP registry lets an agent discover the tool through a standard handshake before it knows anything about the tool underneath.

Composition across tools. An MCP server can expose several underlying tools through one endpoint, so an agent has one place to connect rather than knowing about a list of individual CLIs. That is an operational convenience when the number of tools is large.

Those are production infrastructure arguments. They apply when you are running agents at scale, auditing their behavior, and connecting them to a broad surface of tools through a managed layer. The MCP server is, as a label for what it is, an AI-facing API with logging. That is a useful thing to have. It is just not what you build first.

The order of operations

CLI first. It works for every caller the moment it exists. The MCP server is the wrapper you add when the logging and protocol standardization are worth the layer, not before.

For fledge, the CLI is the foundation a human running commands, an agent driving a repo, and an MCP server talking to downstream clients all sit on. The agent story is not “agents use fledge through MCP.” It is “agents use fledge the same way everything else does, because the CLI was built first-class for any caller.”

When an MCP server for fledge exists, it will be a thin layer. The work is already done in the core. That is the point of building the primitive first.

Scaffolding and templates

The second thing that pushed me toward fledge, after the sameness problem, was bootstrapping: the copy-paste-the-same-setup pain I named back in chapter one. Here is what it looks like up close. You decide to make a new little Rust CLI and the first hour isn't the CLI. It's the Cargo layout, the config, the lint setup, the CI, the README skeleton, all the stuff you've typed out twenty times before. That's the tax, and I was paying it constantly because I'm spinning up a lot of projects.

So scaffolding is a pillar, not a side feature. Standing a project up from nothing is part of the lifecycle just as much as building and testing it is. The whole idea is: stand a new project up from a template instead of copy-pasting it together by hand.

In fledge that lives under `fledge templates`. You run `fledge templates init` with a name and a template and it stands the project up for you:

```
fledge templates init my-tool --template rust-cli
```

There's a built-in set covering the kinds of projects I actually start. The ones that ship are `rust-cli`, `ts-bun`, `python-cli`, `go-cli`, `ts-node`, `static-site`, `kotlin-kmp`, and `kotlin-ktor-api`. That list is basically a map of the languages I work in: a Rust CLI, a couple of flavors of TypeScript, a Python CLI, a Go CLI, a static site, and the Kotlin Multiplatform and Ktor API ones. Those eight are the full built-in set in the repo today. The detection side of fledge knows how to handle Swift, Rust, Node, and the rest once a project exists; the templates side is how a project gets to exist in the first place.

The part I care about most is that templates aren't a closed list I have to bless one at a time. You can scaffold from any GitHub repo, not just the built-ins. `--template user/repo` and it pulls the template from there:

```
fledge templates init my-app --template user/repo
```

The core ships a small useful set, and past that you extend it yourself without me having to be in the loop. A template is just a project shape someone already figured out, and if it lives in a GitHub repo, fledge can stand a new project up from it. So the set of templates isn't "what CorvidLabs published." It's "every starting point anyone has ever put in a repo."

And there's a full set of verbs around templates, not just `init`. There's `fledge templates create` for making one, `fledge templates list` and `fledge templates search` for finding them, and `fledge templates validate` for checking that a template is actually well-

formed before you lean on it. If I'm going to scaffold from a template, and especially if an agent is, I want to know the template holds together before it stamps out a hundred projects with the same broken thing baked in.

It checks more than "does it parse." It reads the template's `template.toml` manifest, confirms the name and description aren't empty, and checks that any tools the template says it requires are actually on your PATH. Then it walks every file and every filename in the template and runs the placeholder templating over them, so a bad placeholder (a syntax error, or a variable the manifest never defines) gets caught as an error before you ever scaffold from it. It also flags a template with no files, and warns if the manifest isn't set to be left out of the output. So it's structural, but it goes past "the files are there": it actually exercises the templating the way `init` will, and tells you where it breaks.

Which is the real reason scaffolding belongs in this CLI and not in some separate generator tool off to the side. Same reason as everything else in fledge: it's the same surface, for the same two kinds of users. A person runs `fledge templates init` and skips the boring hour. An agent runs the exact same command, non-interactively, and stands a fresh project up from step one without a human clicking through a wizard. From the first chapter, detect, introspect, maybe scaffold, then run, the scaffold step is the "maybe." When the repo already exists, fledge detects it. When it doesn't exist yet, the first real move is standing it up from a template, and then everything else, the build verb, the test verb, the review verb, works on it immediately, because it came out of the template already wired the way fledge expects.

A project that already speaks fledge from birth is the thing I was really after. Not just "save me the copy-paste," though it does that. A scaffolded project builds, tests, and is ready for the same verbs as every other repo, from the first commit. The boilerplate I used to paste in by hand was, half the time, exactly the wiring that made a project consistent with my others. Folding scaffolding into the lifecycle CLI means that consistency is the default a project is born with, not something I bolt on after.

So yes, a new project starts with the scaffold. And even when I don't reach for `fledge templates init` by name, the project gets the same setup wired in from the first commit anyway: spec-sync, fledge, augur, attest. The real "init" isn't the template, it's the stack going in. The command is just the fast way to get there. Either road, a new project of mine is born already holding the tooling every other one has.

AI review in the loop

The third pillar is the one people are surprised to find in a task runner. Task running, scaffolding, sure, those are obviously lifecycle things. But AI code review? In the same CLI you build and test with? That feels like it belongs somewhere else, in its own tool, in CI, in a bot on your pull requests.

It doesn't, though, and the reason is simple: review is part of the loop, the same way building and testing are. You write something, you check it, you fix it, you go again. Review is the checking step. And if agents are the ones writing the code now, which, for me, they mostly are, then grading that code is just another verb. It sits right next to `build` and `test` because it does the same job they do: it tells you whether the thing is actually any good before you move on.

One surface beats three tools, and it beats three harder for an agent than for me. That's the whole argument for baking review into the lifecycle CLI instead of shipping a separate tool. An agent in a separate review tool has to learn a whole other thing, with its own invocation and its own output shape, to do one continuous loop of work. If the agent already knows how to drive fledge to build and test, then `fledge review` is a verb it already knows the shape of. Same surface, same JSON, same headless mode. No new tool to learn just because the step changed from "does it compile" to "is it good."

In fledge this is `fledge review`, and what it does is AI code review against the default branch. So it's not reviewing the whole world. It's looking at what changed, your diff against the branch you'd merge into, which is exactly the unit review actually happens on. The same scope a human reviewer or a PR bot would look at, run as a verb in the same CLI you already live in.

And it isn't tied to one model or provider. Under the hood, review goes through the same multi-provider client the rest of my tooling uses, corvid-ai, so fledge talks to any provider corvid-ai supports, Anthropic's API, any OpenAI-compatible endpoint, and local runners like Ollama, among others. The list of supported providers lives in the corvid-ai repo and changes as the ecosystem does. The point is that the review runs against whatever model you want, your call, not the tool's.

Worth being blunt about what that means, because the rest of this book is blunt about blast radius: `fledge review` takes your diff, and your specs, and ships them to whatever provider you pointed it at. If that's a hosted endpoint, your unmerged code just left the building. For a private repo that's a real data-egress surface, and it's your call to make with eyes open, not a detail to gloss. Pointing it at a local model is the version where nothing leaves the machine. One honest wrinkle on that local case: the Ollama path is happy to take a key and a Cloud URL, but the corvid-ai README still frames Ollama as the local, keyless option, so the docs read like

Ollama means the box under your desk. The Cloud path works today; the README just hasn't caught up to it. That's a doc gap on my end, not a missing feature.

There's also a multi-model angle I really like. `--with-model` lets you run a panel, parallel critiques on the same diff from more than one model at once.

```
fledge review --with-model ollama:gpt-oss:120b-cloud,ollama:qwen3-coder:480b-cloud
```

That's not a gimmick. If you've spent any time with these models you know they don't all catch the same things, and they don't all hallucinate the same things either. Running a couple of them across the same diff and seeing where they agree, and where one of them flags something the others missed, is a better signal than trusting any single one. It's a second and third opinion, run in parallel, on the exact change in front of you.

And like everything else in fledge, the output is structured. Every command emits `{schema_version: 1, ...}`. JSON by default. So a review isn't a wall of prose an agent has to read back and interpret like a person would. It's data. The agent that wrote the code can run the review, get structured findings back, and act on them in the same loop, without a human in the middle translating "the reviewer seems unhappy about the error handling" into something to actually do. A person can read the review, and an agent can parse it, off the same command.

`fledge review` is also spec-aware, which the spec-sync chapter covers in full; here it's just worth seeing what review does with a spec once it has one. Review figures out which specs cover the files in the diff, matching on the spec's declared files and on the `specs/<name>/` directory, and folds those specs into the prompt as background. And it's pointed at them deliberately: the model is told the specs describe what the modules are *supposed* to do, to use them to interpret the change, to review only the diff and not the specs themselves, and, the part that matters, if the diff contradicts a spec invariant, to call that out as a bug in the diff. So drift from the spec isn't background flavor on the critique; it's a finding the review is explicitly told to surface.

And it earns the spot. `fledge review` has caught a real bug for me. Something that would have shipped, sitting right there in a diff that built fine and passed its tests, that the LLM flagged before it merged. That's the test of whether review-as-a-verb is worth anything, and it passed it: not a style nit, an actual defect the rest of the loop had waved through. The spec-aware side has paid off too. Pointing the reviewer at the specs has caught code that quietly drifted from the contract the module was supposed to honor, drift that compiles and passes just like the bug did. The agents lean on it well, because to them it's the same shape as build and test: run it, read the structured findings, fix what it found, go again.

If agents write the code, run the build, and run the tests, grading the code is one more verb in reach that they already know, and it catches things the others let through.

Plugins, and not trusting them

fledge has a small core that does little, with all the real capability in plugins. That's on purpose. If the core had to know about every language, every workflow, every team's weird step, it would rot. So it doesn't. The core knows the common verbs and how to detect a project; the actual reach comes from plugins dropped in around it, and anyone can add one without touching the core.

The other reason to push capability out into plugins is that I didn't want to own the list of what fledge can do. People extend it however they like: Rust, Swift, TS, shell. You write the plugin in whatever you're comfortable in. You're not forced into my language to extend my tool.

The way a plugin talks to the core is a real, versioned contract: a binary with a `plugin.toml` manifest, talking JSON with fledge, the same whether it's native or a sandboxed WASM module. The protocol chapter has the wire format and the capability handshake; here it's enough to know the seam is a contract, not a pile of conventions.

And to head off a question that comes up: plugins and lanes are not the same thing. A plugin adds a capability, a new verb. A lane chains verbs you already have into an ordered pipeline. So it's not "everything is a plugin." There's a real core with the three pillars built in, plugins extend the capabilities around it, and lanes string those capabilities into sequences. I give lanes their own chapter later.

Any language, which means you can't trust them

The flip side of "extend it however you like, in any language" is that you end up running code you didn't write and can't vouch for. A plugin is just somebody else's program. Once you've decided anyone can write one in anything, you've also decided you're going to be running a lot of untrusted code.

So fledge sandboxes plugins with WASM, on Wasmtime. The point is safety. A plugin can't read your whole disk or phone home unless you allow it. By default it's boxed in: it gets what you grant it and nothing else.

There's a second reason for the sandbox, and it's the real one. Agents run these plugins. If agents are installing and running plugins, you can't trust them by default. Not the plugins, and not, blindly, the decision to run them. An agent reaching out, grabbing a plugin, and executing it is exactly the situation where "it's probably fine" is not good enough. The sandbox makes agent-run tooling safe. That's the bridge to the trust and blast-radius stuff I get into in the agents book; here the tooling-side version is simple: untrusted code plus an automated thing running it equals you need a box around it. WASM/Wasmtime is the box.

The choice of WASM over the obvious alternatives is worth naming. The two you'd reach for first are seccomp profiles (Linux syscall filtering) and containers (Docker or similar). Both work, but both add friction or assumptions that don't fit here. seccomp requires OS-level privilege to configure and is Linux-only, so it breaks the cross-platform promise immediately. Containers mean a Docker daemon, a separate image pull, and a heavier process boundary than "run a plugin." WASM fits differently: Wasmtime embeds directly in the fledge process as a library, ships inside the single binary, runs on any OS without additional daemons or privileges, and enforces the capability boundary structurally at link time rather than through a kernel policy someone has to configure. It's not that seccomp or containers are wrong. It's that a tool shipping as one binary to run anywhere needs a sandbox that travels with it, and Wasmtime does.

The canary

A sandbox you can't prove is just a hope. So there's a canary, a plugin whose job is to try to do the things a sandboxed plugin should not be able to do, and confirm it can't. It's the proof the sandbox holds. If the canary can't break out, the boundary is real; if it ever could, you'd know immediately.

It's actually two plugins. `fledge-plugin-canary` is the native one. It runs unsandboxed and proves the attacks *work*: reading SSH keys and AWS creds, pulling env vars like `GITHUB_TOKEN`, opening network connections, spawning processes, writing into `.git/hooks`. Then `fledge-plugin-canary-wasm` runs the same battery inside the Wasmtime sandbox, where every one of those should come back BLOCKED. Its own description is blunt about it: it "proves the Wasmtime sandbox blocks every attack the native canary exposes."

The point is sharper when you see the two side by side instead of taking my word for it. The native canary's own output contrasts itself against what a WASM plugin running the same code would see:

```
NATIVE: ~/.ssh/ READABLE            → WASM: BLOCKED (no preopened dir for ~)
NATIVE: ~/.config/fledge/ READABLE   → WASM: BLOCKED (outside sandbox)
NATIVE: GITHUB_TOKEN LEAKED          → WASM: BLOCKED (not passed to guest)
```

Same attack, two boundaries. Native reads your SSH key; WASM can't, because there's no preopened directory for it to reach `~` through. Native inherits `GITHUB_TOKEN`; WASM can't, because the guest's environment is empty. Any result that comes back LEAKED instead of BLOCKED on the WASM side means the sandbox escaped. The two together are the end-to-end check: one shows the danger is real, the other shows the box holds.

What the sandbox does not protect against

The sandbox is blast-radius containment. It bounds what a plugin's code can reach. It does not bound everything, and selling it as a cure for problems it doesn't solve would be the dishonest version of this chapter. So here's the part WASM doesn't buy you.

It does not stop data egress higher up the stack. The WASM box keeps a plugin from reaching your SSH key, but it doesn't govern data you hand a tool on purpose. `fledge review` ships your diff and your specs to whatever LLM provider you pointed it at, and if that's a hosted endpoint, unmerged code just left the machine. No WASM boundary touches that, because the data leaves through the front door, not the plugin. That's a consent-and-policy problem, and I treat it as one in the review chapter, not something the sandbox solves.

It does not protect against a plugin that runs as the host user. Not every plugin is WASM. Native plugins run with your permissions, your environment, your disk. That's the whole reason the native canary can read your SSH key: it isn't sandboxed. When you run a native plugin, or one that forks a process out from under the guest, you're trusting it the way you trust anything you run on your machine. The sandbox is the WASM path's guarantee, not a blanket one over every plugin.

And it does not validate origin or intent. WASM contains what code can do. It says nothing about whether the code, or the agent that wrote the code the plugin runs, should be doing it. A plugin executing agent-written logic is still executing logic I didn't review, inside the box. Untrusted code in a box is still untrusted code. The box just keeps the damage from spreading.

So the honest framing is narrow: WASM/Wasmtime is origin-agnostic blast-radius containment for the code execution path. It is not data-flow control, not consent management, and not a reason to stop thinking about what you hand a plugin or who wrote it.

The plugin ecosystem is bigger than I'd have guessed if you only knew fledge by its three native built-ins: many `fledge-plugin-*` repos in the CorvidLabs org across several languages (the ecosystem chapter covers them), written in whatever language fit the job. The sandbox is what lets that be true: a plugin can be written in anything and still be safe to run. I do the full tour of the ecosystem and the languages in its own chapter.

How fledge fits into the day

Agents running untrusted plugins at speed across many repos is why the sandbox isn't negotiable. The thing making those calls isn't usually me deciding case by case. It's agents, running constantly, across a lot of repos. The WASM sandbox underneath is what makes that safe to let run.

The plugin protocol

This is the seam, the actual place where a plugin and the core meet. It has a name in the repo: the protocol is versioned, and the version string is `fledge-v1`. A plugin declares it, and that's the handshake. Everything else hangs off it.

What a plugin author actually writes

A plugin is a git repo with a manifest at the root called `plugin.toml`, plus one or more executables. The manifest is short. You name the plugin, give it a version, say which protocol you speak, and list the commands you're adding:

```
[plugin]
name = "fledge-deploy"
version = "0.1.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[capabilities]
exec = true
store = true
metadata = false
```

That's the whole contract on the author's side. Each `[[commands]]` entry is a new verb fledge will know about, pointed at a binary to run. The `[capabilities]` block is the author saying, up front, what this plugin needs to be able to do: run shell commands, persist a little state, read project metadata. Default is `false`. You ask for what you need and nothing more, and the asking is visible in the manifest, not buried in the code.

How the host talks to a plugin

When you invoke a plugin command, fledge spawns the binary and talks to it over stdin and stdout as JSON lines, one JSON object per line. The first thing the host sends is an `init` message, and that message is the zero-config idea made literal. It hands the plugin the whole situation: the project name and root, the detected language, the git state (branch, dirty or clean, remote), the plugin's own version and directory, the fledge version, and the exact capabilities that were granted. The plugin doesn't have to go discover where it is or what it's looking at. The host already knows, that's the core's whole job, so it just tells the plugin.

After that it's a conversation. The plugin sends messages back: `prompt`, `confirm`, `select` to ask the user something; `log`, `progress`, `output` to report; `exec` to run a command; `store` and `load` for its little patch of state. Anything with an `id` gets exactly one answer back: a `response` or a `cancel`. Stderr is never captured, so a plugin author can always print debug straight to the terminal.

It's plain enough to read over someone's shoulder. The host opens with `init`, handing over the situation:

```
{"type": "init", "protocol": "fledge-v1",
 "args": ["staging"],
 "project": {"name": "my-app", "root": "/Users/dev/my-app", "language": "rust",
   "git": {"branch": "main", "dirty": false, "remote": "origin"}},
 "capabilities": {"exec": true, "store": true, "metadata": false}}
```

The plugin, already knowing where it is, asks fledge to run a command:

```
{"type": "exec", "id": "6", "command": "git tag -l 'v*' --sort=-v:refname", "timeout": 10}
```

And the host answers the matching `id`:

```
{"type": "response", "id": "6", "value": {"code": 0, "stdout": "v0.9.1\nv0.9.0\n", "stderr": ""
```

That's the whole shape: `init` delivers everything an agent would otherwise have to guess at as structured JSON instead of prose in a README, and every request with an `id` gets exactly one matching response. The tool tells the plugin where it is; the plugin doesn't hunt.

Where the sandbox bolts on

Everything above describes a native plugin, a normal executable, talking JSON over pipes. The trouble, and the repo is blunt about this, is that a native plugin runs as you, with your full access. The capability block gates the protocol, but it doesn't gate the *process*. A plugin that asked for nothing could still read your SSH keys, because it's just a program running as your user. That's the leak the canary proved: declaring capabilities at the protocol level was security theater, and it's the whole reason for the WASM move. I told that arc in the sandbox chapter; here the point is just what it changed in the protocol.

WASM plugins keep the exact same `fledge-v1` protocol but change the boundary. The plugin declares `runtime = "wasm"` and ships a single `.wasm` binary. Instead of stdin and stdout, the same JSON messages cross through three host functions (`recv`, `send`, `exit`) that the host links in. Same message types, same conversation. And the capabilities stop being a polite request. They're enforced at link time: if you didn't grant `exec`, the `exec` import simply isn't

linked, and a plugin that tries to call it fails to instantiate at all. There's no runtime check to fool, because the function it would call doesn't exist for it. Filesystem access is `none`, `project`, or `plugin`, mounted as preopened directories; network is a boolean. On top of that the runtime is fuel-bounded and memory-capped, so a plugin can't spin forever or eat the machine.

So the capability you see in `plugin.toml` and the capability the code can actually reach are the same thing, structurally.

The reason it had to be structural is the lesson the canary taught: a capability you merely *declare* is a capability you're trusting the plugin to respect, and the whole reason you're sandboxing a plugin is that you don't trust it. The only version that holds is the one where the capability you didn't grant isn't reachable, where the function literally isn't there to call. When agents are the ones installing and running these things, that's what you want.

One version risk is worth naming here. The protocol string a plugin declares is `fledge-v1`. When a breaking protocol change comes, and it will at some point, that string becomes `fledge-v2`. A plugin that still declares `fledge-v1` against a host expecting `fledge-v2` fails at instantiation, loudly, before any code runs. It doesn't silently misbehave. The version in the manifest is what forces that early failure: a mismatched plugin doesn't reach the point where it can do anything, so the operator knows exactly what needs updating rather than chasing a subtle runtime bug.

Lane lifecycle hooks

The same protocol carries a second kind of integration: lifecycle hooks. A plugin can register a hook in its `plugin.toml` that fires on `lane:pre` or `lane:post` events instead of on a direct command invocation. The deploy plugin is the example that ships with the docs: it registers a `lane:post` hook that runs automatically after any fledge lane completes.

When the hook fires, fledge passes context to the hook binary through environment variables: `FLEDGE_LANE_NAME`, `FLEDGE_LANE_STATUS`, and `FLEDGE_LANE_RUN_ID`. The lane definition has no knowledge of which plugins are installed. The plugin has no knowledge of which lanes exist. The lifecycle event is the seam.

Here is what the full picture looks like. The lane lives in `fledge.toml`; the hook registration lives in the plugin's `plugin.toml`; the two wire together without either knowing the other's details:

```
# fledge.toml
[lanes.verify]
description = "Pre-merge gate: format, lint, test, build"
steps = ["fmt", "lint", "test", "build"]
```

```
# plugin.toml (from fledge-deploy, or any plugin registering a lane hook)
[plugin]
name = "fledge-deploy"
version = "0.2.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[[hooks]]
event = "lane:post"
binary = "bin/fledge-deploy-hook"

[capabilities]
exec = true
store = true
metadata = false
```

When `fledge lanes run verify` completes, fledge fires the `lane:post` event. Any installed plugin with a matching `[[hooks]]` entry gets its hook binary called with `FLEDGE_LANE_NAME=verify`, `FLEDGE_LANE_STATUS=success` (or `failure`), and `FLEDGE_LANE_RUN_ID` set.

The lane emits structured context as environment variables rather than prose on stdout, so a plugin reads a status it can act on instead of scraping text and guessing. That is the same design principle as the `init` message: the host tells you what happened; you do not have to infer it.

The three pillars sit on top of this

The three pillars (task running, scaffolding, AI review) are core, not community plugins. The protocol is the extension layer *around* them. The spine is fledge's, and `fledge-v1` is how anyone bolts more capability onto it. The core is real and the plugins ring it; the protocol is just the seam where the two meet.

The plugin ecosystem

There are dozens of `fledge-plugin-*` repos in the CorvidLabs org, across several languages, with a handful archived. The live count is real, but it's the least interesting thing about the ecosystem. Plugin count is inventory. The argument is which ones I'd actually hand you, and how sure I am about each.

So here are the five you'd use daily: `augur`, `attest`, `format`, `gitleaks`, `deps`. Start there. The rest of this chapter is about why the other thirty-seven exist and how much to trust them.

The five you'd reach for

These run on basically every change, mine and the agents'. They cluster into two jobs.

The trust pair is `augur` and `attest`. `augur` scores a diff's change risk, proceed, review, or block, from structural signals alone, no API key and no LLM, and its description says the audience out loud: "for humans and agents." `attest` records the signed provenance of who reviewed which commit, kept in git notes. Those two are the ones I lean on hardest, because they're the ones that matter when an agent is the author. Risk scoring on the way in, a provenance trail on the way out.

The dev-loop three are boring on purpose: `format`, `gitleaks`, `deps`. Run the formatter. Scan for committed secrets. Check dependency health, outdated, audit, licenses, across Rust, Node, and Python. None of them is exciting. All of them run constantly, because that's exactly the kind of check you want firing automatically instead of remembering.

If you only ever installed those five, you'd have most of what I get out of the ecosystem.

Three tiers, and how sure I am

The honest way to read the roster is by confidence, not by count. The plugins fall into three tiers, and I trust them very differently. Check the live roster in the CorvidLabs org for the current list; what follows is how to read whatever you find there.

Core. `augur`, `attest`, `format`, `deps`, `gitleaks`. The daily five. I wrote these, I run them on every change, and they're maintained and tested. This is the tier I'd stake the workflow on.

Integration. `github`, `discord`, `algochat`, `memory`. These wire fledge into something external: the GitHub API through `gh`, Discord webhooks for CI-failure notices, encrypted on-chain messaging, a memory store. They're maintained and tested too, but they carry the risk of whatever they talk to. An integration is only as reliable as the service behind it.

Playgrounds. `weather`, `roast`, `hangman`, and the rest of the one-offs. `weather` prints a terminal forecast. `hangman` plays hangman with identifiers pulled out of your codebase. `roast` runs a commit through the LLM and roasts it, "entertainment purposes only." These are not maintained to the same bar. They exist to prove the sandbox works: if a half-serious weather plugin written on a whim is safe to run, the protocol is doing its job. The low bar isn't a weakness. It's the proof.

Core and integration are maintained and tested. The playgrounds prove the sandbox holds. Don't read the flat count as an even spread of equally vetted tools, because they aren't, and pretending otherwise would be the dishonest version of this chapter.

Provenance, plainly

I wrote most of these. The daily five are mine. So are nearly all the one-offs, written on a whim because publishing a plugin costs nothing once the protocol exists. I needed a thing, I wrote a plugin, I dropped it in. The point of the protocol is that I don't have to grow the core to add a capability.

I'm not going to claim a community of outside authors I don't have. A few plugins came from the circle, but I'm not presenting the roster as community- vetted breadth. It's mostly one person filling a drawer, and that's the honest provenance.

The canary, and why any of this is safe

One pair earns its own mention because it's load-bearing for the whole ecosystem: `fledge-plugin-canary` and `fledge-plugin-canary-wasm`, the red-team pair from the sandbox chapter. The native one proves the attacks work. The WASM one proves the sandbox blocks them. Everything else here, a plugin written in Kotlin, a plugin I wrote half-asleep, is only safe to run because that pair holds the boundary. The ecosystem can be chaotic precisely because the box around it isn't.

Why so many languages

Most are Rust and shell, then Swift, TypeScript, Kotlin, Python, and a few HTML/JavaScript ones. That spread is the protocol working as intended. The whole reason for `fledge-v1` and the WASM sandbox is that a plugin can be written in anything and still be safe to run, so the ecosystem doesn't have to agree on a language, only on a contract. `fledge-plugin-bridge` is Kotlin. `fledge-plugin- memory` is TypeScript. `fledge-plugin-attest` is Swift. Half the integrations are shell. Nobody had to learn Rust to add to fledge. They wrote the thing in whatever was in front of them, and the core stayed the same size the whole time.

I don’t own the list of what fledge can do. The protocol does, and it lets anyone add to it without asking me. The spread is the proof, and the tiers are the honesty about it.

spec-sync and staying honest

Spec drift is the failure mode I care about more than most. You write a spec, the contract for a module, what it does, what its public API is, and then the code drifts. Somebody changes a function, the spec still claims the old shape, and now the document and the code quietly disagree. With a human team that's a stale README. With an agent in the loop it's worse, because the agent reads the spec as truth, builds on a contract the code no longer honors, and nobody notices until something breaks. spec-sync exists to make that drift impossible to ignore.

spec-sync is its own tool, its own Rust binary, its own GitHub Action, and its tagline says what it is: "Bidirectional spec-to-code validation with cross-project references, dependency graphs, and AI-powered generation." The two words that matter are *bidirectional* and *validation*. It checks, in both directions, that the spec and the code agree. It is not a test suite and it is not a fuzzy diff against prose. It's structural contract checking: does the documented public API actually match the real one. The two directions are not symmetric: an export the code has but the spec doesn't document is a warning, something to fill in; an entry the spec claims that the code doesn't have is an error, a broken promise.

What it actually checks

A spec is a markdown file, `*.spec.md`, with YAML frontmatter and a set of required sections. The frontmatter names the module, a version, a status, and the source files it covers. The required `##` sections are Purpose, Public API, Invariants, Behavioral Examples, Error Cases, Dependencies, and Change Log. Miss a required section and the spec is incomplete and validation blocks.

That's the rule; here's a spec honoring it. This is the head of fledge's own `review` module spec, the real frontmatter, and the `## Public API` section filled in with the actual exports spec-sync checks against the code:

```
---
module: review
version: 10
status: active
files:
  - src/review.rs
db_tables: []
depends_on:
  - spec
  - llm
  - config
```

```
---

# Review

## Purpose

AI-powered code review of current branch changes...

## Public API

### Exported Functions

| Export | Description |
|--------|-------------|
| `run` | Entry point for the review command |
| `ReviewOptions` | Options struct: base, file, json, model, provider, with_model |
| `ReviewFormat` | Enum: Summary, Checklist, or Inline |
```

The `files:` line is what spec-sync intersects against the diff and against the real exports of `src/review.rs`. Every row in that `## Public API` table is a name spec-sync expects to find in the code; an entry the code doesn't have is the error case below. If you've never written a spec, that frontmatter plus one honest `## Public API` table is the shape to copy. The rest of the required sections are the same kind of thing, prose and tables describing what the module promises.

Then it validates both ways:

- **Code → spec:** undocumented export, warning.
- **Spec → code:** phantom or stale entry, missing source file, type mismatch, all errors.

It does the same for schemas: declared database tables and columns get checked against the real SQL, and a phantom table or column fails. And it resolves references across projects, so a spec in one repo can depend on a module in another via `owner/repo@module` and that link gets verified too.

The thing to hold onto is that this is *structural*. It's not grading your prose and it's not generating tests. It's asking one blunt question, does the documented surface match the real surface, and answering it deterministically. That's what makes it safe to put in front of an agent. There's no judgment call to argue with; either the API matches the spec or it doesn't.

How it shows up

It shows up three ways, and in the interview the answer to "how does spec-sync show up day to day" was: all of them. Three front doors onto the same spec format, each running somewhere different:

Front door	Where it runs	What it does
`fledge spec` (native)	your machine, inside fledge	`init`, `check`, `list`, `show`: fledge's own validation, no shell-out
`specsync` binary	the agent's loop, before a PR	`specsync check` / `--fix`: the standalone tool the agent runs in-loop
`CorvidLabs/spec-sync@v4`	CI, on the pull request	gates the build, comments drift, blocks on failure

The rest of this section is those three rows, one at a time.

fledge knows specs natively. spec is one of fledge's pillars. The fledge README is verbatim about it: "Spec | `spec` | [spec-sync]. Modules declare their contract, AI uses it as context." Worth being precise about what that means in the code: fledge has its own `fledge spec` (`init`, `check`, `list`, `show`) built right into the binary. It doesn't shell out to the `specsync` tool. It reads the same `.specsync/config.toml` and the same `*.spec.md` files, parses them itself, and does its own check. So the spec format is shared, but the validation inside fledge is fledge's own code, not a call out to the separate binary.

That native spec-awareness is also why fledge's AI commands can lean on the contract. `fledge ask` is spec-aware Q&A and `fledge review` is spec-aware code review, both pull the relevant specs in as context. The spec is the context those commands feed the model. The spec isn't a document off to the side; it's the thing the tooling reads when it reasons about your code.

It gates CI. There's a standalone GitHub Action, `CorvidLabs/spec-sync@v4`, that runs `specsync check` automatically. A minimal workflow:

```
- uses: CorvidLabs/spec-sync@v4
  with:
    strict: 'true'          # warnings become errors
    require-coverage: '100' # minimum spec coverage
```

`strict` turns warnings into errors. `require-coverage` sets a floor. `comment` posts a spec-drift summary onto the pull request. And the check exits non-zero on failure, which means the PR is blocked. That's the mechanism that makes the whole thing real: drift doesn't generate a polite note, it fails the build. An agent, or a human, can't quietly let the spec and the code part ways, because the part-ways *is* the failing check.

It keeps the agent on-spec. This is the one I care about most, and it's why the spec lives inside the agent's loop, not just in CI. The mechanics are concrete. The agent runs `specsync check` as part of its loop, the same way it runs build and test. The check comes back with structured failures, and the agent reads them and decides what to do, and which way it goes depends on which direction the drift runs.

Here is what that structured output looks like when both directions have drifted (illustrative example matching the real output format):

```
specsync check

CHECKING src/review.rs against review.spec.md
  WARNING  undocumented export: `ReviewOptions::with_model`
             → run `specsync check --fix` to add stub

  ERROR    phantom export: `ReviewFormat::Detailed`
             spec claims this variant; code has: Summary, Checklist, Inline
             → update spec or add the variant to code

  ERROR    phantom export: `run_async`
             spec claims this function; not found in src/review.rs
             → update spec or add the function

SUMMARY  2 errors, 1 warning
EXIT 1
```

That's what the agent reads. The errors name the file, the spec claim, and what the code actually has. The warnings name the export and the exact fix command. No guessing required: either reconcile the code to match the spec or correct the spec, then re-run until exit 0.

If the code grew an export the spec doesn't mention, the warning direction, the agent doesn't hand-edit the spec. It runs `specsync check --fix`, which auto-adds the undocumented exports to the spec as stubs, and the easy case is reconciled in one move. The agent's job there is mostly to fill the stub in with a real description, not to discover the drift in the first place.

The other direction is the one that takes actual work. When the spec claims an API the code doesn't honor, the error direction, the stale or phantom entry, there's no `--fix` for it, because the fix is a judgment call: either the code is wrong and the agent goes and makes the code match what was promised, or the spec was aspirational and the agent corrects the contract. Either way the agent has to reconcile it in the code, deliberately, and re-run the check until it's green. That's the loop in practice: contract, change, check, and then either an auto-add or a real correction depending on which side fell out of step, and it runs before the diff ever reaches a pull request, inside Merlin's loop, not after the fact in CI.

What spec-sync does not check

There's a line I want to be honest about, because it's the edge of what this tool does. spec-sync checks that the spec and the code agree. It has no opinion on whether the spec is any good.

Think about what that leaves open. A spec can name the right exports and describe them wrong. It can be thin, a Purpose section that says nothing and a Public API table that's accurate and tells you nothing about what the module is for. It can be aspirational, written for the code somebody meant to write. And if an agent drafted the spec from a task brief that was itself wrong, or injected, the spec can be a faithful contract for the wrong thing. In every one of those cases spec-sync passes. The surface matches the surface. The check is green. The contract is wrong, and nothing in the loop caught it, because checking the spec against the code can't tell you the spec was bad to begin with.

That's a real gap and I'm naming it as one. The code has a check; the spec doesn't. What you'd want is a second gate that runs on the spec itself, before an agent ever builds against it: does every required section actually say something, does the Public API point at files that exist, is there a clear statement of what success and failure look like, and did a human actually sign off on this contract. That's a `fledge spec lint`, and it isn't built yet. Until it is, the spec is the one input to this whole pipeline that nothing validates, and that's worth knowing when you trust the green check.

Why a shared contract is the whole point

This ties straight back to the thesis under all of these tools: humans and agents using the same tools, as equal first-class citizens. A spec is the clearest version of that idea I have. The human writes or reviews it; the agent builds against it; spec-sync holds both of them to it, in both directions, deterministically. The human who refactored without updating the doc gets caught the same way the agent that hallucinated an API gets caught. The check doesn't care which of them drifted.

Building fledge in Rust

fledge is a single binary. You install it once and it runs on any machine, any OS, with nothing else to install alongside it. No runtime to pull in, no interpreter, no dependency manager that has to already be present. Cross-platform by default: one build pipeline produces a binary that works on macOS, Linux, and Windows. That's the core requirement, and Rust is the reason it's easy instead of a fight. The whole chapter follows from that: why Rust was the right call, why picking it up wasn't the war story people expect, and how agents covered the fluency gap.

I'd been writing Swift for about a decade when I started fledge. So the honest version of this chapter is a little anticlimactic: picking up Rust didn't hurt. Nothing major fought me. Not even the borrow checker, which is the part everyone warns you about.

I think people expect a war story here: the language that humbled me, the month I spent losing fights with the compiler. I don't have one. And I'd rather tell you why it went smoothly than invent the drama.

Swift prepped me

A lot of Rust felt familiar because Swift had already trained me on the same ideas, just wearing different clothes.

Enums and pattern matching were the big one. Swift's enums are real sum types, and I'd been leaning on them and `switch` for years. Rust's `enum` and `match` are the same muscle. Result and Option weren't new either. I'd been living in Swift's optionals and `Result` for a long time, so "this can be a value or nothing" and "this can be a value or an error" were already how I thought about code. Rust just makes you handle both, all the time, out loud. That wasn't a new discipline for me; it was a discipline I already had, now enforced.

Traits landed as roughly Swift's protocols. Not identical, but close enough that I wasn't learning a foreign concept. I was learning a dialect of one I knew. "Define behavior, conform types to it" is the same shape in both.

So the language didn't feel like a wall. It felt like a place I'd half lived before. The value-type and optionals habits Swift drilled into me are, I think, exactly why the borrow checker never became a fight. If you already think in terms of ownership and "who holds this value," the checker is mostly telling you things you were already trying to do. It wasn't a new way of thinking I had to acquire. It was a stricter referee for a game I already knew how to play.

And the shape of the thing it builds is the simple one I wanted: fledge is a single Rust binary, built with Cargo. One crate, one binary out the other end.

Agents did the heavy lifting

I'm not going to pretend I'm a fluent Rust programmer. I'm not. I'm a fluent Swift programmer who can read and steer Rust well, and that's a different thing.

What closed the gap was agents. I leaned on them hard to be productive in a language I'm less fluent in. The familiar parts I could read, reason about, and direct on my own. I knew what I wanted the code to *do* and what good structure looked like, because that part transfers across languages. The parts where Rust has its own idioms, its own way of saying a thing, the agents carried.

This is a distinction worth drawing, just seen from the other side. My Swift is handwritten. My Rust is agent-amplified. In one I'm the author at the keys. In the other I'm the one who knows what right looks like, pointing agents at it and checking their work.

And honestly, building fledge in Rust this way is a small proof of the whole thesis behind it. The tool is built for humans and agents both. It got built *by* a human and agents both. The agents that drive fledge today helped write fledge in the first place.

The tooling was a relief

Here's the part I didn't expect to enjoy as much as I did: Rust's tooling felt like a relief.

Cargo just works. One tool for building, testing, dependencies, the whole thing, and it's the same everywhere. The crate ecosystem is deep. Whatever I needed, there was usually a solid crate for it, and pulling it in was one line. And single-binary builds are exactly what fledge wanted to be: I needed to ship one light binary that runs everywhere, and Rust hands you that by default.

The contrast is with Swift tooling once you step off Apple's platforms. On Apple, the Swift story is great. Off it (Linux, cross-platform, the kind of "runs anywhere as one binary" target fledge needed) it gets harder. That's a real reason fledge is Rust and not Swift, even though Swift is my home language. I wanted a light, single binary I could drop on any machine, that agents could run anywhere, and Rust's toolchain made that the easy path instead of the fight.

So that's the builder's-eye truth of it. Swift prepped the thinking, agents covered the fluency I don't have, and the tooling (Cargo, the crates, the single binary) was the part that actually made me glad I'd switched. Picking the language was the easy decision. The work that actually took thought was everything else: figuring out what fledge should be.

How I use it, and who uses it

Let me lead with the honest part instead of saving it for the end. fledge is not widely used. It's mine, it's my agents', it's my circle's. That's the point. The tool got built to solve the builder's actual problem first, and it does, every day. A tool that solves the problem in front of you beats a tool with a logo wall and no dog in the fight.

So the question this chapter answers isn't "how many people use fledge." It's "who, and why that's the right order." The answer isn't "everybody," and it was never supposed to be.

My agents drive it more than I do

The thing people get wrong when they picture how I use fledge is that they picture *me* using it. That happens, but it's not the main way fledge runs anymore. My agents drive it, by a wide margin. When an agent is working a repo, fledge is how it builds, tests, and runs what it just changed: its execution surface, not just mine. That's exactly what the first chapter said the design was for. Most days the agents get more mileage out of it than I do, and I'm mostly steering.

That's the user base I built for. Not a crowd of strangers. Me plus the agents, in every repo I have, all the time.

Who else uses it

fledge is open source, it's on the Homebrew tap, anyone can `cargo install` it. None of that is an adoption claim. A tap is a distribution channel, not a user count, and I'm not going to dress one up as the other.

Today the user base is me, my agents, and the CorvidLabs circle, the people I actually work with. There's no broad external adoption. No community of strangers filing issues. It's personal and circle infrastructure, and that's worth saying plainly instead of implying a groundswell that isn't there.

The collaborators matter to the picture, not just as a headcount. fledge being the shared surface across the circle is part of the design. When someone in CorvidLabs picks up one of my repos, they don't have to learn that repo's private dialect. They already know the verbs, because the verbs are the same everywhere. The same consistency that saves me relearning lands the people I work with on a surface they already know.

Should you use it

Maybe not yet. Honestly, only if you work like I do: a lot of repos, agents in the loop, one consistent surface across all of it. If that's your problem, fledge solves it. If it isn't, the tool is overkill and I'd rather tell you that than sell you on it.

What I won't claim is that limited adoption proves the design works at scale. It doesn't. It proves the design works for me, which is a smaller claim and the only one I can stand behind. fledge earns its keep inside my own world first, every day, mostly through agents, with me steering and a small circle on the same surface. That's a tool doing its job. The breadth can come later or never. The job is already getting done.

Where fledge goes

The honest answer to “where does fledge go next” is less exciting than people expect, and I think that’s a good sign. It’s mostly mature. The big shape of it is built. What’s left is mostly to maintain it and keep dogfooding it. There are a couple of directions I’d grow it in, but I’m not sitting on a grand roadmap of reinvention, because fledge doesn’t need reinventing. It needs to keep being the thing everything else rides on.

A bigger ecosystem

The playground and integration tiers are open: anyone can add to them without asking. The core tier is not. That asymmetry is intentional and it is not changing.

The direction with the most room is the one the ecosystem chapter already described: more plugins. fledge gets more capable by the ring around the core growing, not by the core fattening: a richer set of plugins so that whatever repo you drop fledge into, there’s already something that knows how to handle it. More integrations, more of the little one-offs that each solve one real thing.

Tied to that is broadening what fledge detects and runs out of the box, the zero-config promise from the first chapter, held in more places. Every language and platform it doesn’t yet know is a repo where the same-verbs-just-work promise has a gap, so filling in detection and platform coverage is the other obvious direction. Not glamorous. Coverage is just what makes a universal surface actually universal.

And the coverage that matters most right now isn’t the next language. It’s the operating system. fledge has to run first-class on all three: Windows, Linux, and macOS. A tool that’s only really at home on one OS isn’t a universal surface, it’s a local one. So cross-OS support is the next real frontier, more than adding another language detector.

What it means to maintain something agents depend on

I could list features here and promise a reinvention. That would be the wrong frame for what fledge is now.

The question that matters in 2026 isn’t what features fledge adds. Code is cheap. Agents write most of it. What’s scarce is a tool you can actually trust to run your work. An agent can generate new functionality faster than I can review it. What it can’t generate is a tool with years of dogfooding behind it, a stable interface, behavior a pipeline can count on.

A tool with years of dogfooding and a stable interface a pipeline can count on is what fledge is, and it is the harder thing to build. Not harder technically. Harder because it requires deciding that stability is the output, not a byproduct. Every tool starts life as a new feature. Very few survive long enough to become infrastructure. The ones that do are the ones where the maintainer treated “nothing broke” as the main accomplishment, not a consolation prize for a slow sprint.

I use fledge in every repo, all the time. My agents drive it constantly. That means the bugs find me. The missing verb, the detection that guessed wrong, the plugin hook that fired in the wrong order. I hit it because I live on it, and I fix it because I live on it. The dogfooding is not a side activity. It is the whole quality mechanism. Maintaining and using are the same job.

The introduction to this book asked what makes a tool worth depending on. The answer it pointed at was: one clean surface, no hidden interactive steps, a real plugin boundary, a spec that stays honest as the code changes. Build that, and an agent can drive it on day one, because it was never built to need a person in the first place.

fledge is that. Not because of any single design decision but because of the accumulated result of every time I refused to let a shortcut erode the interface. The verbs are the same across every repo. The JSON is the same shape. The `fledge introspect` manifest stays current. Nothing ships that I haven’t run on my own work first.

When agents are the ones operating most tooling, what earns trust isn’t a long feature list. It’s showing up the same way every time, for long enough that something real got built on top of it. That’s a quieter ambition than a roadmap slide, and it’s the one that matters.

So the job from here is the job it’s always been: keep it solid, keep living on it, keep the surface honest. I’ll keep hitting the bugs because I keep running it, and I’ll keep fixing them. That’s the whole plan.

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.