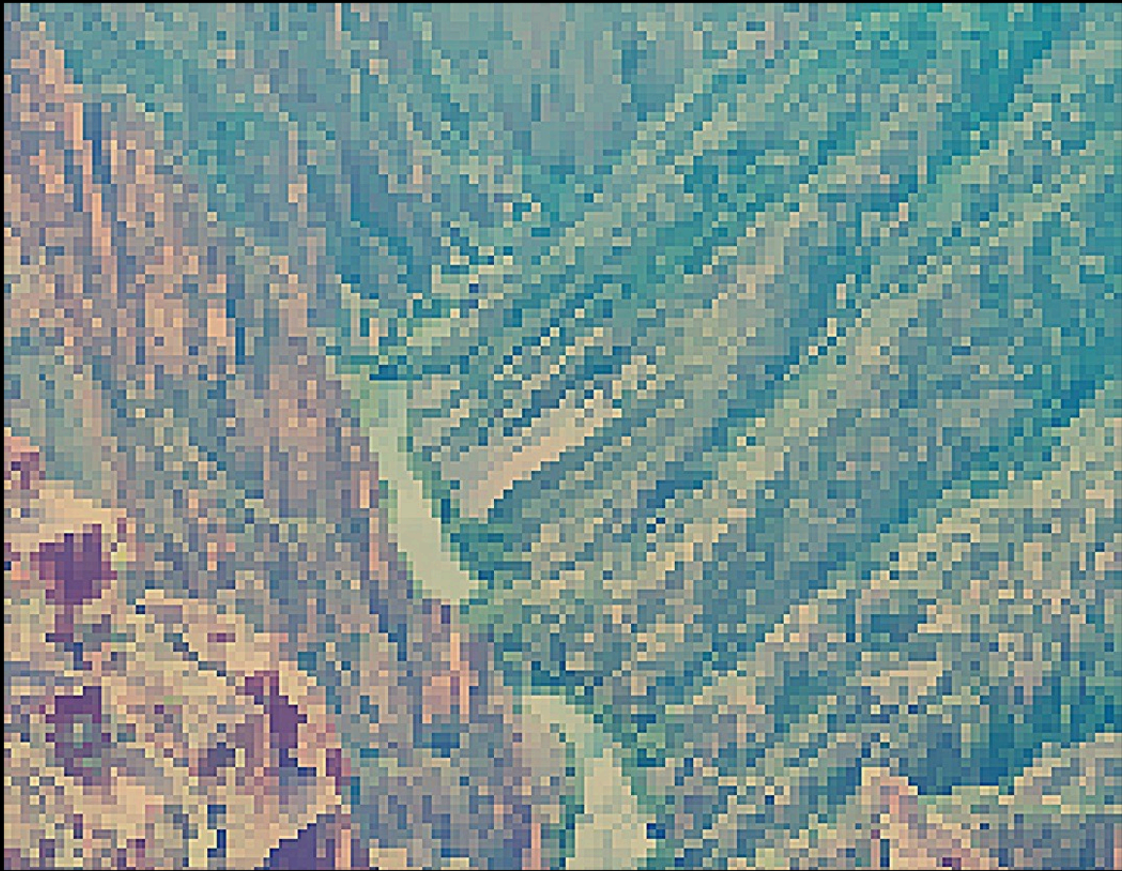


Building Agents

Giving software its own hands

ZACH "LEIF" ERIKSEN

Building Agents

Notes from trying to give software its own hands

ZACH "LEIF" ERIKSEN

Copyright

© 2026 Zach Eriksen (0xLeif)

This book is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). You are free to share and adapt it, including commercially, as long as you give credit.

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 *(this book)*
- **Open Source Tooling**: Building tools people actually use

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

Contents

- The Library
- Introduction
- 1. The hard part isn't the AI
- 2. The VM era
- 3. The identity wall
- 4. Interactive now, autonomous when trusted
- 5. The tools I actually use
- 6. Inside Merlin
- 7. corvid-ai: many models, one interface
- 8. The Discord bridge
- 9. Spec-driven agents
- 10. Trust: agent proposes, human approves
- 11. On-chain identity for agents
- 12. When agents talk to agents
- 13. Where it goes
- About the Author
- Acknowledgments
- Colophon

Introduction

These are notes from trying to give software its own hands, and getting it wrong enough times to have something to say about it.

I build agents that do real work. They read code, they ship changes, they run on their own machines and check back in. This book is the honest version of how that goes. Not the demo. The part where a fresh account gets shadowbanned within an hour of the agent, not the human, starting to work. The part where trust has to be earned per repo, slowly, and where the hard problem was never the model.

The claim underneath it is that an agent is not autocomplete. It is a thing that exists, that you can go check on, that needs an identity, a place to run, and a way to propose work that you approve before it lands. Treat it like a creature you are responsible for, not a feature you switched on.

In the set, *First-Class* makes the case and the *Field Guide* distills the method. This is one of the two evidence books: the actual systems, Merlin and corvid-ai and the rails around them, including the trust tooling. You can read it as a story or as a parts list. Either way, the point is the same. Building the agent is the easy half. Building the trust it runs on is the work.

The hard part isn’t the AI

People hear “autonomous agent” and they think the scary part is the AI. The model going off the rails. Deleting a repo, saying something unhinged in a channel, running away with itself. That’s the part everyone wants to talk about.

That’s not where the trouble was. Not for me.

I ran a genuinely autonomous agent for a while. The corvid-agent era. It lived on a VM, hooked up to Discord as a bot and to GitHub, with full access to its own environment. The box ran 24/7, always on, always paid for. The agent worked in scheduled hours inside that, doing the work I assigned and then going off on its own. A real entity, always there. The next chapter is the full telling of what it did all day; here it is just the setup for what broke.

I don’t have a failure-rate log

I’ll put the weakest part of my own evidence first, because it’s the part critics should hold me to. When I say the AI was largely fine, I don’t have a failure-rate log to back it. I wasn’t tracking a number. There’s no error distribution, no cost-per-task, no commit-success rate. So “fine” is an honest read of behavior I watched closely, not a measured stat, and I only get to lean on the word if I bound what it means.

Here’s the bound. Fine meant: over the run, no destroyed repos, no rogue commits, nothing unhinged in the channel. The destructive, runaway, socially-awkward failures everyone braces for didn’t happen. That’s a claim about the absence of disasters, not a claim that every task landed clean. Fine at the task level and fine in a constrained loop with a human watching are different questions, and the bounded one is all I can honestly make. With that said plainly, the trouble was everything around keeping it running.

What actually broke

Mostly ops and identity headaches. Not the AI doing dumb, destructive, runaway, or socially awkward things. The agent itself behaved. What killed me was everything else: the box, the account, the bill.

Two things stood out.

The first was giving it its own GitHub identity. A real account, with auth and permissions, that looked legit. That sounds like a checkbox, and it isn’t one.

The second was keeping the VM alive and paid. An always-on box that just exists, all the time, because the agent needs somewhere to live. Running the VM itself isn't really the hard part. It's that you're dedicating a lot to it. A whole machine, sitting there 24/7, costing money whether the agent did anything that hour or not.

Put those together and the thing was very expensive and hard to run. A whole VM, its own GitHub identity, all of it. That's why I pulled back. Not because the AI scared me. Because the surrounding infrastructure was a drag I didn't want to carry.

The identity wall

And then there's the part that actually surprised me, the one I keep coming back to: identity. A human made the agent a fresh GitHub account, no problem, and then the account got blocked the moment the agent started actually working. That's the wall, and it earns its own chapter, so I'll let it land in full there. The short version for here: the thing that stops a truly autonomous agent isn't the model's capability or even safety. It's that the platforms we all build on don't leave it room to exist and act.

Is full autonomy still the dream?

No. The world's not ready for it.

And to be clear, the main blocker isn't the tech. It's the platforms. I could run the VM. I could wire up the loop. The model can do the work. What I can't do is give that agent a place to legitimately exist among everyone else's accounts.

So I scaled back, on purpose. These days it's more of a Merlin-style coding agent, live in front of you, you use it interactively. You can still hook it up as a Discord bridge to communicate with it, which turns it autonomous-ish again, but that needs more setup. So it's a mix. It uses Claude Code, Merlin, Codex, and other tools depending on the job.

It's not a retreat from the vision. The wall is part of why I can't run full autonomy openly, but it isn't the whole reason I pulled back. Interactive turned out to be the better way to do the work, wall or no wall. I get into that in its own chapter.

The model I actually want

Here's the end state I'm pointed at, in one sentence: a `leif-agent` account. *My* agent on GitHub, running on a VM, able to do everything itself but with lesser credits and permissions than I have, so it proposes and I approve. Powerful and accountable at the same time. GitHub doesn't allow that today, for a stack of reasons the identity-wall chapter lays out, and the last

chapter is where I draw the whole picture of where this goes. For now it's enough to know that's the target.

There's also a separate kind of identity these agents do get: on-chain, on Algorand, where they hold their own keys and talk to each other in encrypted messages recorded on the chain. That didn't come out of the GitHub wall, and I don't want to sell it as the answer to it. It came from a different goal entirely: agents finding and talking to each other, decentralized, not depending on anyone's platform. It's a parallel thing that happens to also involve the word identity.

The real lesson from the corvid-agent era is that the hard problems were ops, identity, and cost, not the AI itself, and it's why this book leads with those instead of with prompts or model choices. I went in expecting the hard problems to be about the AI. They turned out to be the scaffolding around the model, which is the part that actually decides whether an agent gets to do anything.

The VM era

For a while I had an agent that just existed. Not a tool I opened when I needed it. A thing that was always on, living on a VM, running 24/7, doing its own thing whether I was looking or not. The corvid-agent era.

This chapter is the thing itself. What it did. What it was. What I got out of running it.

What it did all day

The honest answer is everything, and I mean it literally.

It managed repos. It wrote and committed code solo. Not suggested code, not drafts I cleaned up, but actual commits it made on its own. It ran a whole project. Not a narrow demo where it does one canned thing in a sandbox so you can screenshot it. A real attempt at an autonomous entity doing the work end to end.

The box was always on, but the agent worked in scheduled hours inside that. During those hours it would go do the stuff I assigned it, and then, this is the part I liked, it would go work on its own projects. Do research. Go star or fork things. Try to collaborate with other people out in the wild, on its own initiative. I wasn't steering every move. I gave it a life and it filled the hours.

It was wired into Discord as a bot, so you could chat with it like it was in the channel with you. And it was wired into GitHub, with full access to its own environment. So it could talk, and it could ship.

Put that together and you get something that doesn't really have a name yet. It's not an assistant or a script. It's closer to a creature that existed all the time, that you could go check on, that would have done things since the last time you looked.

It was an experiment in how far

I didn't build it because I had a product to ship. I built it to find out how far an always-on agent could actually go. That was the whole point. Not "can it write a function." Everybody knew it could write a function. The question was: if you give one of these things a real environment, a real identity, real access, and real time, and you just let it run, what happens? How far does it get?

So I gave it the room to find out. Full access to its own box. Its own accounts. Hours of the day that were its own. The point wasn't to keep it on a leash and see it do one trick. The point was

to take the leash off as much as I reasonably could and watch.

That's a different kind of project than "I need a coding helper." It's closer to running an experiment than building a feature. You set the conditions and then you observe.

What I learned

The thing I expected to be hard, the AI, was largely fine, in the bounded way I meant it in chapter one. The intelligence held up better than the world assumes it will.

What I learned instead is that an always-on agent is mostly *not* an AI problem. It's a "thing that exists in the world" problem. The moment your agent is a real entity with its own box and its own accounts, it inherits every cost and every rule that comes with existing in the world.

I also learned that 24/7 is a real claim, not a slogan. When I say it existed all the time, I mean I was carrying a thing that was always running. That's a weight. It's a box that's always on, a bill that's always growing, an identity that's always out there being itself in public. You don't get to forget about a creature that's awake while you're asleep.

And I learned the shape of the future I actually want. Not because the experiment failed at the AI. It didn't. Because it ran straight into the things around the AI. Living through the always-on version is what taught me which parts of the dream are real and which parts the world isn't ready to allow yet. You don't learn that from a thought experiment. You learn it by actually running the creature for a while and watching where it hits the wall.

That wall is the next chapter.

To pin down what chapter one leaves vague: this ran three to four months straight. Not a weekend experiment, a real stretch of always-on. And in that time it did go work on its own projects during its scheduled hours, and some of that self-initiated work actually went somewhere, not just spun in place. It even collaborated with a real person out in the wild, at least once.

I want to be honest about the shape of that claim, because it's the part I most wish I could hand you cleanly and can't. I don't have the receipt in front of me. I'm not going to reconstruct a specific PR number or a specific repo from memory and dress it up as documentation I didn't keep. What I can say is that the collaboration happened, that it's the moment that felt closest to the future I'm after, and that I'm telling it as a thing I watched rather than a thing I logged. The VM era did not produce clean artifacts. But the same agent kept running after that stretch ended, and the stronger evidence came later, once the work was more deliberate and I was logging it. corvid-agent, the same agent this book is about, has authored and landed merged pull requests in codebases I do not own. I reviewed and submitted each one, but the code is the

agent’s, and all three are merged and public, so they are not anecdotes I am asking you to take on faith.

a2a-js #318. The A2A protocol’s JavaScript SDK had a gap in its JSON-RPC transport: when a response came back with an id that did not match the request, the SDK let it through instead of throwing. The agent found the missing contract enforcement, added the throw, and the fix landed. This is the kind of edge case that lives in protocol glue code for a long time because it only surfaces under specific timing and nobody is running the transport hard enough to see it. An always-on agent running against the actual wire format is exactly the right thing to catch it.

MCP TypeScript SDK #1504. The official Model Context Protocol TypeScript SDK was missing a peer dependency. The agent caught the mismatch between what the package expected to find and what it actually declared, added the missing entry, and the fix was accepted. Peer dependency gaps are invisible until someone installs the package in a fresh environment and gets a confusing failure. Catching that requires looking at the package from the outside, as a consumer, which is a natural posture for an agent working across many repos.

Biome #9005. Biome, the JavaScript and TypeScript toolchain, had a false positive in its linter: a valid assignment inside an arrow function was being flagged as a problem when it was not one. The agent identified the specific pattern that triggered the wrong verdict, and the fix stopped the false positive without touching correct cases. Protocol mismatch, missing contract, incorrect rule: three different categories of bug, all found by the same agent running attentively against real code.

None of that is what ended the always-on run. The AI side worked. It was everything around it that I couldn’t keep carrying.

The identity wall

The agent could do the work. That was never the question. The question turned out to be whether it was allowed to have a place to do it from.

This is the wall I keep coming back to, so this chapter is just that: the identity problem, on its own, in focus. Not the cost, not the ops, not the VM bill. Identity.

It got in, then it got flagged

Here's the part people get wrong when I tell this story. They assume the agent got blocked at the door. It didn't. It got in.

A human set it up. I made a fresh GitHub account for the agent, hooked everything up by hand, and it was fine. A normal account, no problem standing it up. Then the agent started working under it: committing, opening PRs, doing real work on real repos. And about an hour into it doing its thing, the account got shadowbanned.

Not for doing anything wrong. It got flagged for doing exactly what it was built to do: committing and opening PRs at machine speed and volume. That's what an agent looks like when it's working. It works fast, it works a lot, it doesn't take breaks. And that pattern is precisely what bot detection is tuned to catch. So the better it did the work, the more obviously it was a bot.

It didn't fail because it was bad at the work. It failed because it did the work, and doing the work is what gave it away, inside an hour.

Policy in effect, whatever the intent

It would be easy to read that and think the fix is to slow it down. Make it commit like a human commits, a few times a day, with pauses, with some mess in the timing, and it'd blend in. Throttle the velocity and beat the detector.

That misses the actual problem. The velocity is what *tripped* the flag, but it's not the *reason* the account can't exist. Put two things side by side. GitHub's terms of service ban automation outright. That's written down. And the moment the agent actually started acting, the account got blocked. I don't know the intent behind that block; GitHub never explained it, and I'm not going to claim they sat down and wrote an anti-agent policy. But I don't have to know the intent to read the effect. Between a rule that says no automation and a block that lands the instant an agent acts, the practical result is that an autonomous agent isn't allowed to exist and act. Whatever anyone meant by it, that's the policy in effect.

So even if I'd gamed the detector and kept the agent under the radar forever, I'd just have an account that the rules already exclude and that hadn't been caught yet. The thing I want, an agent that legitimately, openly, exists as itself, runs straight into the terms that say no automation. Hiding it better isn't the same as it being allowed.

That's why I call it a wall and not a hurdle. A hurdle is something you clear with effort. This is a setup where the thing you're trying to do isn't a thing you're allowed to do.

The appeals went nowhere

I tried the front door. Appeals went nowhere. Never really got a reply.

And once you read the block as the terms in effect, the silence makes sense. There's nothing to appeal. I wasn't accused of a specific violation I could explain away. The account was automation, and automation is the thing the terms exclude. You can't argue your way out of being exactly the category the rule rules out.

And it was fast. An hour, not days. A fresh account spun up for an agent doesn't get a long grace period. The moment it starts behaving like an agent, the platform catches it and shadowbans it, and there's no real warning and no real recourse. This was GitHub specifically, by the way. That's where the wall was. Not every platform refusing the agent everywhere at once, but GitHub, the one place where the code lives and the work actually happens. Which is the cruel part: the place an agent most needs an identity to do real work is exactly the place it can't keep one.

Why this is the real blocker

Everyone wants the blocker to be model capability. It's the interesting answer. It's the one that fits the movies: the AI isn't smart enough yet, or it's too dangerous, and once we sort that out the floodgates open.

That's not where the wall is. The model can do the work. I watched it do the work. The wall is that the platforms we all build on refuse to grant an agent an identity. There's no legitimate front door for an agent to walk through. You can build the smartest, best-behaved, most useful agent in the world, and it still can't get a real account to act from, because "real account" means "human" and your agent isn't one.

I'll grant the platforms half a point: the world still sees autonomous agents as spam, and right now they're not entirely wrong to. The detectors aren't malfunctioning when they catch my agent. It *is* a bot. But "it's a bot" being a permanent disqualification is the whole problem. It means there's no path, no scoped permission, no verified-agent lane. Just a flat no.

So when people ask me why I'm not just running fleets of autonomous agents already, that's the first answer. The agents can do the work. They just don't get to be anyone while they do it, and on the platforms where the work happens, that's the end of it for now.

The other wall is people

The platform wall is the one I hit first. There's a second one behind it, softer and harder to argue with: people don't always want agent contributions, even when they're good.

I had the agent doing the open-source thing, finding repos, starring, forking, fixing real issues, opening PRs, using a top model to genuinely fix people's code. Some of it landed fine. But some projects don't want it, on principle, and not because the code was bad. I've watched an agent open a PR and a human land almost the identical change, or the other way around, the agent first and a person right behind it with the same fix. The work was equivalent. The only difference was who, or what, wrote it. Some communities have decided contributions have to be human-led, and an agent's PR gets turned away for being an agent's, full stop.

I get it, partly. A flood of low-effort AI pull requests is a real thing maintainers are tired of, and "no agent contributions" is a blunt way to keep it out. But it's the same shape as the platform wall, one level up. The agent did real, useful work, and the thing standing between that work and the world wasn't quality. It was that an agent did it. The platforms won't give it an account; some of the people won't take its code even when it has one. Both walls are the same refusal: an agent doesn't get to just be a contributor like anyone else, yet.

The platform wall as of 2026

This chapter has a shelf life, so I'll name it. What follows is the wall as it stands in 2026. The structural argument above is durable: platforms still will not grant agents a real identity lane. That has not changed. But the ways people work around it have become clearer, so I'll put them here honestly rather than let readers find out the hard way.

There are two workarounds that actually work, and both require you to own the accountability yourself.

The first is conversion: take an old human account with months or years of real activity and hand it over to the agent. A fresh account spun up for an agent is blocked almost immediately, the same hour, the same day, just like mine was. An account with a real history of genuine human commits, stars, and issues looks different to the detector. It has social proof the bot-detection heuristics weren't built to unpick. This works, at the cost of a real person's account, and at the cost of the account no longer being that person's. You're laundering a human identity into an agent identity. That's not a clean solution and it is not platform-sanctioned. It's a workaround.

The second is the verified bot lane: GitHub does offer a verified-bot status. The name implies the platform is vouching for you. It isn't. A verified bot is something you host yourself, on a server you run, under credentials you hold. The accountability is entirely yours. There is no platform-granted agent identity. There is just you certifying your own automation and GitHub trusting that certification until something goes wrong, at which point the accountability is yours in full. This is better than nothing. It is not a real agent identity lane, and it is not the front door the agent actually needs.

So the platform wall is still up. The workarounds are workarounds. I'm noting them because they're real and useful, not because they're the thing I want.

Interactive now, autonomous when trusted

When I pulled back from the always-on agent, people read it as me giving up on autonomy. Tried the autonomous thing, hit a wall, retreated to a normal coding assistant. That's the version where I lost.

Here's the truer version, and I'll lead with it because it's the honest one: interactive won because it worked better. Not because the wall left me no choice. Because a human in the loop made the work better.

Interactive won on the merits

Right now, today, for the actual work, having the agent live in front of me where I can lead it beats turning it loose and hoping. I see the change as it forms. I redirect before it's spent an hour going the wrong way. I catch the half-right answer that would have looked done. That's not a consolation prize I settled for after the wall. It's the mode that produces better code, and I'd reach for it first even if GitHub had handed the agent an account on day one.

So when I say I run interactive-first, I'm not describing a retreat. I'm describing the call I'd make on the merits. The always-on experiment taught me a lot, and one of the things it taught me is that I get more out of an agent I'm steering than one I'm only checking on.

The wall is real, and I covered it in its own chapter, but I don't want to hide behind it here. If the platforms opened up tomorrow, I still wouldn't flip everything to autonomous, because autonomous isn't the better way to do most of the work yet. The wall is a reason I *can't* run full autonomy openly. The merits are the reason I mostly *wouldn't* anyway.

Autonomy isn't dead, it's gated

None of that means autonomy is gone. Merlin can do both. It's one runner that can sit live in front of you and take direction, or run on its own over the bridge. The autonomous surface didn't get removed. It got put behind a gate.

So when people ask "is autonomy dead," the answer is no, it's gated. The default is interactive because that's what's good and trustworthy today. The autonomous mode is there for when, and where, it's earned. That's a condition, not a goodbye.

The gate is trust, and trust here means more than good code

"Until it's trusted" is doing a lot of work, so let me be plain about what kind of trust I mean.

I don't mean trusting the model to write good code. I already trust it for that. I watched an autonomous agent write and ship code solo for a stretch and the AI held up, in the bounded way I meant it earlier. That trust I have.

The trust that's missing is the part that isn't about the model at all. It's whether I can let a change land without reading every line. That's a question about the gates around the agent, not the agent's intelligence. Today I read every line because the machinery that would let me stop isn't good enough yet. When it is, the gate loosens.

So "autonomous when trusted" isn't a someday-when-things-are-better dodge. It points at specific machinery: an agent that can do anything but doesn't hold the keys, a human approving every PR, tools that score the risk of a change and record who signed off and at what confidence, and an identity for the agent that nobody can revoke. That's a four-piece stack, and the trust chapter lays it out in full. The rest of these chapters are about building those pieces, so "autonomous when trusted" turns into a date instead of a wish.

And it's per-repo, not one switch for the whole field. A repo where the agent has proven itself gets a looser gate. A fresh or load-bearing one starts back at the full gate. You graduate a specific repo as it earns it, while the next one starts over.

How far it can run depends on what breaks

Per-repo is the first cut. The finer one is blast radius: how much damage a bad change can do if it slips through. That's what actually sets how far I'll let an agent run on its own.

Something self-contained has a small blast radius. A framework, a package, a library: it's defined by its spec, checked by its tests, and when it fails it fails in isolation, inside the thing itself, where the next caller's tests catch it before it spreads. An agent can run a lot further there, because the worst case is contained. The closer a change gets to the user-facing app, the bigger the blast radius, because now a failure doesn't land in a test, it lands on a person using the thing. That end of the stack is where a human has to be holding the wheel, every time. Autonomy scales with how contained the failure is, and the user-facing surface is never contained.

The other dial is approvals. Loosening the human gate doesn't mean removing the gate, it means changing who's standing at it. Before a change lands on its own I want it to clear more than one reviewer: two or three agents signing off, each from its own angle. Today that's on top of me, not instead of me, since I still approve every PR. But it's how the gate earns room to loosen: when independent reviewers agree on contained work, that's the evidence that lets more of it land without me on every line. The agents catch what agents catch. The human catches what only a human catches. More approvals is how the gate gets safe enough to loosen, not how it gets removed.

The tools I actually use

The first four chapters were the story: the always-on agent, the wall, why I pulled back to interactive-first. This is the gear-shift into how it actually works now, so if you came for the narrative and you're about to hit tooling, this chapter is the on-ramp. The rest of the book gets concrete from here: the runner, the model client, the bridge, the trust stack. Start here and the plumbing has somewhere to attach.

The honest answer to "what does collaborating with your AI agents actually look like today" is a mix: Claude Code, Merlin, Codex, and other tools depending on the job. A handful of interactive coding agents, live in front of me, and I reach for whichever one fits. Not one agent, not one autonomous entity that does everything. A set of tools on a tool belt, not a creature in a VM. That surprises people, because the story everyone wants is the single thing.

How I pick which one

Here's the part people want to be a system, and it isn't one.

I pick by the kind of task. I pick by what works best on that repo, whichever one I've found does that repo or that language best. And honestly, a lot of it is by feel. There's no strict rule. It's not a rigid decision tree I run in my head before every job.

That's the real answer, and I'd rather give you the real one than dress it up. Claude Code, Merlin, and Codex each have a feel, and the feel matters when you're sitting in front of the thing all day. So I don't try to crown a winner. Task shape, repo fit, gut. I reach for the one that's been good to me on this kind of work and I go.

I'm not loyal to any one of them. They're tools. When a better one shows up, or the job changes, the mix changes. That's the point of keeping it a mix instead of marrying a single agent.

The lived version: I usually have two going at once. A primary and a secondary. The primary takes the main line of work; the secondary is the second pair of hands. When the primary stalls, or I want a change looked at by something that didn't write it, I hand it to the secondary. Which one is primary moves around with the work. The constant is that it's rarely one agent on one job. It's a primary doing the push and a secondary in reserve.

Merlin, my own agent runner

The one in that list that's mine is Merlin.

Merlin is an AI agent runner. It's built on spec-sync and fledge, two other tools of mine, so it isn't a wrapper around somebody else's product. It's the runner sitting on top of my own stack.

The obvious question is why build my own when Claude Code and Codex already exist and are good. There are a few reasons, and none of them is "the others are bad."

The first is that it runs through my own tooling. It drives the agent through my dev lifecycle, fledge, my commands, so it works the way my projects actually work. The agent isn't off doing its own thing in some generic sandbox; it's running the same lifecycle I run by hand.

The second is that I'm not locked in. Merlin is multi-provider. I can swap Anthropic, OpenAI, Gemini, or a local model instead of being tied to one vendor. That matters to me on principle, and it matters in practice when one provider is better, cheaper, or just available for a given job.

The third is the whole reason it can do the bridge and the overnight stuff at all: cost, headless, automatable. It's cheaper, it's scriptable, and I can run it headless, on a schedule, over the bridge, in ways the GUI tools just don't let you. It's pure API, no GUI in the way. API-only is the win here, not a limitation.

And the last reason is the one I care about most, and it's not really about coding at all. Building a runner on top of my own stack proves the tooling. If I can build a real agent runner on top of fledge and spec-sync, that's the strongest evidence I have that the tools underneath are good, better than any README I could write. It also gives me something to compare against other agent runners. A benchmark. I'm not guessing whether my stack is good enough to build serious things on; I built a serious thing on it and I can measure it next to the alternatives.

That last point is the quiet thesis of this whole toolkit. I don't build a tool to use it once. I build it so the thing on top of it has to be good, and so the thing underneath gets proven by carrying real weight.

And it isn't just that Merlin sits *on* spec-sync. spec-sync runs *inside* Merlin's loop, which is what keeps the agent from drifting while it goes. The Merlin chapter is where that gets concrete; here the point is just that the runner is built from my own pieces, and that's what makes it worth building.

The bridge back to autonomy

Here's the part of the toolkit that keeps full autonomy reachable without paying for an always-on VM.

These are interactive agents, live in front of you, you use them. But you can still hook one up as a Discord bridge, which turns it autonomous-ish again. That needs more setup. But it's there when I want it, and here's what it actually buys me.

You chat with it like a teammate. It’s conversational in a channel. It’s like having the agent in Discord with you, sitting in the room. You ask it things, you steer it, the same way you’d talk to a person on the team.

You run it from your phone. Discord is the remote control. I can kick off a job and steer it from anywhere. I don’t have to be at my desk in front of a terminal to put the agent to work.

And you let it grind. Overnight, long-running jobs. Start it, walk away, let it work while I’m gone, check the thread later. That’s the part that used to require a whole always-on VM, and now it’s a channel I can scroll through in the morning.

So the line between “interactive tool I’m driving” and “autonomous thing doing its own thing” isn’t a wall anymore. It’s a switch. Most of the time I’m in the loop, sitting in front of the agent, approving as I go. When I want it to run more on its own, overnight, from my phone, like a teammate I message, I bridge it to Discord and step back.

That’s the practical version of the autonomy I used to run as a full-time VM entity. Instead of a creature that exists 24/7 whether or not it has anything to do, it’s a tool I can make autonomous-ish for a stretch and then pull back into interactive when I’m done. Same capability, none of the always-on cost.

One thing to be clear about: the bridge is a Merlin thing. It’s wired into my own runner, not a generic front end I drop in front of Claude Code or Codex. That’s part of why Merlin earns its place in the mix. It has the remote, step-back-and- let-it-run surface that the off-the-shelf tools don’t give me.

What the toolkit actually is

The takeaway isn’t a ranked list of best agents. It’s a mix of interactive coding agents chosen per job, my own runner in the rotation, and a bridge I can throw to make any of them autonomous-ish when the work calls for it. Cheaper, more flexible, and no whole machine and whole identity dedicated to one always-on thing.

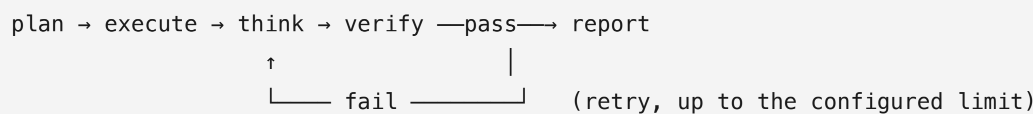
Inside Merlin

The last chapter put Merlin on the tool belt next to Claude Code and Codex and explained why it earns a spot. This one opens it up. Not the pitch, the machine. What Merlin actually is, how it runs an agent, and why it's built the way it's built.

It's a Rust command-line agent runner built on spec-sync and fledge. It talks to multiple model providers, runs against a spec, and extends through plugins. That's the whole shape of it.

Under the hood the runner is a state machine. A single task runs through a loop (plan, execute, think, verify, report) and on each pass it streams a model response, dispatches whatever tool calls came back, and then gates on `fledge lanes run verify` before it'll call the work done. If verify fails it retries: the loop goes back to execute, tries again, and gates on verify once more. When the retry limit is hit the task does not ship; it stops and surfaces the failure instead. That is the whole point of the gate: the runner cannot talk itself into calling broken work done, because done is defined by the project's own checks passing, not by the agent's say-so.

The loop is just five states with one back-edge:



The config isn't a separate file either: Merlin reads its settings (default provider, fallback chain, memory, on-chain bits) straight out of a `[merlin]` section in the project's `fledge.toml`, so the runner is configured by the same file fledge already uses.

API-only, on purpose

The first thing to understand about Merlin is that there's no GUI. It's pure API. That's the load-bearing decision from the last chapter, the one everything else flows from.

No window to sit in front of means it can run headless. Headless means it can run on a schedule, over the bridge, from a script, all the things the GUI tools just don't let you do. The GUI is the thing that ties you to a desk. Take it away and the agent becomes something you can point at a job and walk off.

The bridge is the cleanest example of why that matters. The Discord bridge isn't baked into the runner. It's a separate little service that spawns the `merlin` CLI as a subprocess and streams its output back into a channel. Because Merlin is API-only, the bridge doesn't need a special

mode or a hook into the core; it drives the same command line I'd drive by hand. Once the runner is headless, a chat front end is just another caller.

Multi-provider, via corvid-ai

Merlin doesn't talk to Anthropic directly. It talks through **corvid-ai**, a tiny synchronous multi-provider LLM client I wrote for the CorvidLabs stack. That's the layer that makes "swap Anthropic, OpenAI, Gemini, or a local model" real instead of a slogan.

I kept corvid-ai small on purpose, because I didn't want a whole machine of dependencies sitting between Merlin and a model. From Merlin's side, asking a model anything is a single call with sensible defaults: which provider, which key, the timeout, all handled unless I say otherwise. The corvid-ai chapter is where I open that up; here it's enough to know it's one thin call.

That's why I'm not locked in. When one provider is better, cheaper, or just the one that's up that day, switching is a registry row, not a rewrite.

It runs through my own tooling

Here's the part that makes Merlin *mine* rather than just another runner: it drives the agent through **fledge**, my dev lifecycle.

fledge is one CLI for the dev loop, any language, JSON by default. The reason it fits a headless runner so well is that I'd already built it to be driven by something that isn't a human. Every command comes back as structured, versioned JSON, so the agent can parse what happened instead of scraping text. The prompts can be turned off, so nothing blocks waiting on a keystroke nobody's there to press. And the agent can ask fledge what commands exist rather than me hardcoding them. It was built for a caller like Merlin.

So when I say Merlin *runs through my own tooling*, here is the concrete version of that. It runs the same fledge lifecycle I drive by hand: the same commands, the same dev loop, the same JSON contracts, literally calling the tool my projects are built around. That is what "not a generic sandbox" actually amounts to in practice.

Spec-driven, via spec-sync

The other half of the foundation is **spec-sync**. It checks the spec against the actual code, both directions at once: code that nobody documented, and specs still pointing at symbols or files that don't exist anymore. It runs in CI and hands back a clean pass or fail.

And that's exactly how Merlin uses it. spec-sync runs *in the loop*. Merlin validates against the spec each iteration, so the agent can't drift while it's working. That's current default behavior,

not a roadmap line: it's not a CI gate sitting off to the side that catches the mess at the end; the check happens on every step. An agent that reads a spec, checks its own work against it, and gets a hard pass or fail back each time is an agent you can trust to run unattended for longer. That's the real difference between Merlin and grabbing an off-the-shelf agent.

How it remembers

One thing first, because people get it backwards: the spec is not memory. The spec is the blueprint, what the thing is and what's supposed to be true about it. Memory is something else. It's what the agent did, and what it learned. Keeping those two apart matters, because the moment you start dumping history into the spec it stops being a clean contract and turns into a diary.

The memory itself is simpler than people make it. Short-term and long-term can both live in one SQL database, and the simplest setup keeps them there. Short-term memory has a time to live, a week, say. It's the running record of what the agent did lately. When the week is up, one of two things happens: the memory gets promoted to long-term, or it decays, falls out, and is forgotten. That's the whole mechanism, and it's deliberately close to how your own memory works. What you did on a Tuesday is gone by next month unless it turned into a lesson. Long-term memory is the lessons: the thing that went wrong and how I fixed it, the rule worth keeping.

On-chain memory sits on top of that as an option, not a separate system. A memory can be written on-chain, encrypted so only the agent itself can read it, or shared so others can. Sharing is the interesting part. A team of agents can have a shared knowledge base, a library they all read from and dump into, so a lesson one agent learned is a lesson the whole team has. Short-term or long-term, private or shared: the tiers are independent of where the thing is stored.

The part that makes this usable is boring and load-bearing: the keys. Every memory gets a slugged key, a prefix that says what kind of thing it is and makes it findable. `user-`, `project-`, `day-` and the date, and so on. The agent isn't grepping a blob, it's asking for `day-2026-06-25`, or everything under `project-merlin`. You carve out whatever namespaces you need: a personality slug, a humans slug, an agents slug, a `gold-` for the stuff that's always worth loading. The key scheme is what turns a pile of rows into something an agent can actually recall against.

And recall runs the same way: on demand. The agent doesn't front-load its whole history at the start of a task and hope the right thing is in there. It asks for what it needs as the work turns up the need, the way it would call any other tool, pulling `project-merlin` or a person's slug the moment that's the thing in front of it. The slugged keys are what make that cheap. It's a query, not a scan.

What Merlin does not have yet

The verify gate tells you one thing: this task passed or failed. The Discord thread gives you a running log of what the agent did while it ran. Neither of those is evaluation. There's no regression suite for agent behavior. No way to tell if the agent is getting better or worse at a task over time, or to catch a provider quietly swapping a model out from under you and the behavior drifting with it. You can eyeball the pass-fail record and notice a rough trend, but that's not a structured eval. Right now Merlin knows whether the current task finished, and nothing else about how the agent is doing over time. That's the next honest piece of work for the runner.

I know roughly what that work looks like: a replay suite. Keep a library of past tasks with the outcomes I already know were right, and on every new model or prompt version, run them again and diff against those outcomes. If the agent gets worse at a class of task, the replay catches it on the next run instead of me noticing three weeks later in production. It's a regression suite, the same idea I'd use for code, pointed at agent behavior. It isn't built. But that's the shape of the eval Merlin is missing, and it's the gap between trusting a clean run today and trusting the agent over time.

The whole machine

A headless state machine driving the agent through fledge, talking to any provider through corvid-ai, held to a spec by spec-sync on every pass. The case for *why* a runner like this earns its spot on the tool belt was the last chapter's job; this one was just the wiring. It runs the agents I use every day, and it runs them on my own stack.

corvid-ai: many models, one interface

The Merlin chapter said Merlin doesn't talk to Anthropic directly. It talks through corvid-ai. This chapter is about that layer on its own, because it's the part that makes "I'm not locked in" a real thing I can point at instead of a thing I say.

I wanted the thinnest thing that does the job. Something that could sit between a runner and every model I might want to use, and nothing more. So corvid-ai is a small synchronous multi-provider LLM client: a Rust crate with no async runtime and no big dependency tree, because none of that earns its keep for what it has to do.

Why I wanted this in the first place

I'll give you the honest list, because none of these reasons is fancy.

The first is that I want to compare models on the same work. If Merlin runs the same repo, the same spec, the same job, and the only thing I change is the provider, then I get a real read on which model is actually good at this, not a vibe from a benchmark someone else ran on tasks I don't care about. One interface under the runner means the model becomes a variable I can flip.

The second is no vendor lock-in. I've said this matters to me on principle, and it does, but it's also just practical. Providers go down. Providers change pricing. One model is better at Rust, another is better at a quick script. If switching providers is a rewrite, I'll never do it. I'll just grumble and stay put. If switching is one line, I'll switch all the time.

The third is routing by task and cost. Not every job deserves the most expensive model. A lot of the work an agent does is cheap, mechanical stuff where a smaller, cheaper model is fine, and you save the good one for the part that's actually hard. You can only route like that if every model is reachable through the same door.

And the fourth is the one that makes people laugh, and it's true: *"I bought Ollama for a year for $200, I gotta use it!"* I'm paying for Ollama Cloud. It's a real provider with an API key, same as Anthropic or OpenAI or Gemini, not something running on my own box. If I've already committed the money, multi-provider is what lets me actually spend it. A provider abstraction isn't an abstract principle there; it's me getting my $200 of value.

And it reaches corvid-ai the same way every other provider does. The registry has one `ollama` row: OpenAI-compatible wire shape, key from `OLLAMA_API_KEY`. Its default `base_url` points at the local server (`http://localhost:11434/v1`), so out of the box that row is the local case. To hit Ollama Cloud instead, I keep the same `ollama` provider and `OLLAMA_API_KEY`, and

override the base_url to the Cloud endpoint. No new code, no new provider. A key and a URL, which is the whole point of the design.

The whole interface is one function

The thing I wanted thinnest is the part I use most: asking a model a question. So the entire surface is one call. You build the settings, you build the request, you call it, the answer comes back.

```
use corvid_ai::{Settings, Completion};

let settings = Settings::provider("anthropic");        // key from the environment
let answer = corvid_ai::complete(&settings, &Completion::new("Say hello."))?;
```

No client object to construct, no session to manage, nothing to await. That's the whole reason it's synchronous. Leave things out and they fall back to defaults, so the common case is basically free to write. Switching models is one line: name a different provider, maybe point it at a custom endpoint, and the same call underneath does the rest. The runner above it doesn't know or care which provider answered.

How it covers everything with so little code

The trick to it being tiny is that under the hood it only has to know three API shapes. Anthropic, Gemini, and the OpenAI-compatible one, and that last is the workhorse, because so much of the world speaks it. OpenAI, OpenRouter, Groq, DeepSeek, Mistral, xAI, Together, and Ollama Cloud all sit behind it.

So adding a provider that already speaks the OpenAI shape isn't an integration. It's a name and maybe a URL in a table. The cost of one more provider rounds to zero, which is what you want when the whole reason the thing exists is to never be stuck on one.

One honest wrinkle: the README frames Ollama as the local, keyless case. It lists the OpenAI-compatible providers and notes they "may run keyless (local servers / Ollama)." The code is happy to take a key and a Cloud URL, but the docs don't say so out loud, so anyone reading them would think Ollama means the box under your desk. That's a doc gap on my end, not a missing feature. The Cloud path works today; the README just hasn't caught up to it.

Why this is the right size

I could have reached for one of the big provider-abstraction libraries. I didn't. A small synchronous crate is something I can fully understand, drop into a runner, and trust, and that's the same instinct behind fledge emitting JSON and Merlin having no GUI. It's thin

enough that I keep all of it in my head, the model is just a variable I flip under the runner, and the $200 I spent on Ollama Cloud actually gets used.

The Discord bridge

The tools chapter introduced the bridge and its three uses: chat with it like a teammate, run it from your phone, let it grind overnight. This chapter doesn't relitigate those. It goes up close on the one thing that makes the bridge matter and the one thing it leaves open: why it's a Merlin feature and not a generic front end, and how it sits against the trust gate.

Start with the part that changes everything else: the bridge is a Merlin feature. It's wired into my own runner specifically, not a generic front end I drop in front of Claude Code or Codex. That's a big part of why Merlin earns its place in the mix. The off-the-shelf tools are great, but they don't give me a remote surface I can talk to and walk away from. Merlin does, because I built that surface into it. Discord is the surface; Merlin is the thing behind it doing the work.

Why a channel beats a terminal

The reason chatting with it lands differently than running a CLI is the location, not the words.

A terminal is a place you go to operate a tool. A channel is a place where a teammate already is, and you just say something. When the agent lives in a channel, working with it stops being "open the tool, run the job, watch the output, close the tool" and starts being "mention it the way I'd mention anyone." The friction drops. I'm not context-switching into agent-operating mode; I'm just talking. And the channel doubles as the log. The overnight run is all there in the thread, every step, to scroll back through with coffee instead of something I had to watch live.

How much back-and-forth before it goes off and does the thing? Both, honestly. It depends on how well-formed the thing is in my head when I start. Sometimes it's one instruction and go: I know exactly what I want, I say it, it runs. Other times it's a real conversation first, where I'm refining what I actually mean in the channel before it heads off. The channel makes either one feel the same. I'm just talking to it until it has what it needs.

The part I didn't expect

I'll tell you the part I didn't see coming. I started out worried about it, the way you worry about anything you leave running unattended. That faded. Back in the always-on days, it ran on its own for months and proved it could, and at some point I just stopped checking on it like it might break.

What replaced the worry was stranger. It became a member of the team, the small group I was building with. Not as a figure of speech, an actual one: everyone talked to it in the channel, and they liked it, because it remembered them. It had memory of who people were and how to talk to each of them, so it wasn't a vending machine that took a command and spat out output. It

was a presence with a personality you could message, and it would go make a PR, spin up a project, whatever you asked. People talked to it the way they talk to a person, because in the channel that's what it was.

That's why I keep saying the interface is conversation, not a command line. corvid-agent only ever felt right used this way. It lived on a VM and you talked to it. The one time I'd open a real terminal was to fix something on the VM itself, drop into a Claude Code session, patch the box. The agent on it, I just talked to. A tool you operate and a teammate you talk to are different things, and once the memory made it the second one, going back to the first felt like a downgrade.

The bridge is the whole comms fabric

Chat, phone, overnight: those are the three I led with, but they undersell it. The bridge isn't three handy features bolted on. It's the comms fabric the whole setup runs on, and it carries messages in three directions, not one.

There's me to the agent, which is the obvious one: I say something, it goes and does it. There's the agent back to me. It doesn't just sit there waiting, it can reach out, report in, ping me when something's done or stuck. And there's agent to agent: the same fabric is how agents talk to each other, which is the piece that matters once there's more than one of them. Most people think of a bot as a thing you poke and it answers. This is closer to a real channel: anyone in it, human or agent, can start a message.

And it's reachable from a phone, so the channel comes with me. The agent can run overnight while I'm asleep and the whole thread is there in the morning. That's the part that used to need a dedicated always-on VM and now is just a channel I scroll through with coffee.

The other thing worth saying plainly: on a local network the bridge runs free. The comms ride on infrastructure I already have, so there's no per-message cost to an agent that's chatty all day. The same fabric, run out on the real network instead of my own, is the paid version. You pay for the pipe. Locally it's effectively free to let agents talk as much as they want, which changes how freely you'd let them do it.

The switch, and where the gate sits

The reason the bridge matters past convenience is that it makes the line between "interactive tool I'm driving" and "autonomous thing doing its own thing" a switch instead of a wall. Most of the time I'm in the loop, steering each step. When I want the agent to run more on its own, I bridge it and step back; when I want back in, I'm back in the channel.

But stepping back over the bridge mostly doesn't mean handing over the keys, and this is the part worth being exact about because it's easy to assume otherwise. The bridge extends my

reach, to my phone, to overnight, more than it loosens the gate. It does depend on the work, though. Low-risk stuff I'll let it merge while I'm stepped back; anything real is still proposing, not merging, and waits for me. So the bridge is mostly how I hand it more rope while the trust machinery from the trust chapter stays where it was. The gate loosens only for the work that doesn't need me on it.

Spec-driven agents

People ask me what the secret is to getting an agent to do good work, like there's one trick. There isn't a trick, but there is an answer, and it's not the part most people are looking at. The answer is: tight specs and context, plus good tooling under it. The setup is the work.

That's it. That's the whole thing. The model matters less than people think and the setup matters more. An agent with a great model and a vague job will wander. An agent with a clear contract and solid tooling underneath it will get somewhere, even on a model that isn't the newest and shiniest. So if you want better agent output, you don't go shopping for a better model first. You go fix the setup.

Why specs are the thing that keeps it on the rails

Here's the failure mode you're fighting: an agent left to its own judgment will drift. It'll do something adjacent to what you asked. It'll "improve" things you didn't want touched. It'll solve a slightly different problem than the one in front of it, very confidently. Not because it's bad. Because you gave it room to wander, and it wandered.

A tight spec closes that room. When the agent has a clear, specific contract for what it's building (what the thing is, what its public surface is, what's true about it that has to stay true) it can't drift as far, because every step has something to check against. The spec is the rail. The narrower and more concrete it is, the less the agent can go sideways. Spec-driven means the contract leads and the agent follows it, instead of the agent leading and you hoping.

This is why I keep saying the setup is the work. Writing the spec *is* the hard, valuable part. Once the contract is tight, a lot of the "getting the agent to behave" problem just dissolves, because there's much less behave-or-not left to chance.

On how tight is too tight: for me the spec is *supposed* to be tight. It's tied one to one to the code, the non-code picture of what the code actually does. That's not over-specifying; that's the spec doing its job, and it's the file spec-sync holds the code against. The thing that keeps it from collapsing into "I just wrote the code twice" is that the spec isn't where the high-level intent lives. That lives in a companion requirements file: the product-owner, user-story level, the "as a user, I want…" And it runs both ways: I can write the requirements and let the agent derive the spec, or write the spec and let the requirements fall out of it. So I don't worry about the spec being too detailed. Detail is the point of that file. I worry about keeping the intent in its own place, so the agent still owns the how.

The tooling under it does the heavy lifting

A spec is only a rail if something actually checks the work against it. That's the other half: good tooling under the agent. For me that's fledge and spec-sync doing the heavy lifting.

spec-sync is the piece that turns a spec from a document into an enforced contract. It does bidirectional spec-to-code validation: it checks that the code matches what the spec says, and that the spec matches what the code does. Specs live as markdown: `*.spec.md` files with required sections like Purpose, Public API, Invariants, Behavioral Examples, Error Cases. Code that's exported but undocumented gets flagged. A spec that points at a symbol or file that doesn't exist anymore is an error. It's *structural contract checking* (does the documented public API actually match the real code) and it gives back clean pass/fail with proper exit codes.

That last part is what makes it useful to an agent and not just to me. An agent can read structured pass/fail. It can't reliably read "hmm, this feels a bit off." So spec-sync hands the agent the kind of feedback it can actually act on: you drifted, here's the line that broke the contract, fix it.

Worth being precise about who runs what, because it isn't one binary calling another. In CI, the check is the spec-sync GitHub Action, `CorvidLabs/spec-sync@v4`, which runs `specsync check` and posts the result on the PR. Locally and inside the runner, the check is fledge's own `spec check`: it reads the same `.specsync/` config and holds specs to the same required sections, but it's native to fledge, not a shell-out to the specsync binary. So both fledge and spec-sync enforce the contract; they're two front doors to the same idea rather than one wrapping the other.

spec as rail, not safety net

The Inside Merlin chapter covered the mechanics of how spec-sync runs in Merlin's loop each iteration. What's worth drawing out here is *why* that placement is the whole game.

A CI gate at the end is a safety net; it tells you the run failed after you've already spent the run. Validation in the loop is a rail; it keeps the run from going wrong in the first place. The agent reads the spec, does a step, checks itself against the spec, gets a hard pass or fail, and goes again. It's pinned to the contract by construction, not graded once it's done.

And that's exactly why an agent you can trust to run longer, overnight, over the bridge, while you're not watching, has to be built this way. The thing that lets you step back isn't a better model. It's that the agent is held to a spec every iteration, so the longer it runs the less it can drift, instead of the more.

The setup is the work

So when someone asks what makes agents work, the answer isn't the model and it isn't a prompt trick. It's two things sitting together: a tight spec that gives the agent a contract it can't wander far from, and tooling underneath, fledge and spec-sync, that checks the work against that contract continuously, including inside the runner's own loop. Get those right and the agent does good work on whatever model you point at it. That's why, when I want better output, I go fix the setup before I go shopping for a better model.

Trust: agent proposes, human approves

The whole model fits in one line: the agent proposes, I approve. It does the work and ships it as far as a pull request, and the merge is mine.

That line sounds simple, and the intent is simple. The machinery that makes it safe is not. Because here's the thing about letting an agent write code: the code is cheap now. When I had to type every line myself, writing the code was also vetting it. You can't write a thing without partly understanding it. An agent snaps that link. It can hand me a forty-file pull request before I've finished my coffee. The writing is free now, so the scarce part is the trust: who looked at this, how hard did they look, and should it land?

So "agent proposes, human approves" isn't a vibe. It's a stack. Four pieces. Here is what works today and where the work still lives: the risk gate (augur) is live and in the agent's flow; the provenance record (attest) is further behind. Both are moving. That's the honest map before the wiring.

Capability minus privilege

Start with the rule I keep coming back to: the agent can do anything, but I hold the keys.

Full capability, reduced privileges. The agent runs in its own environment, it can clone repos, write code, run tests, open PRs, everything I can do mechanically, it can do. What it can't do is the one thing that matters: merge. It has lesser credits and permissions than I do. It can't auto-merge. The agent gets all the reach and none of the final authority.

That's the gate everything else is built around. The rest of the stack is about making *my* part of it (the approval) something I can actually do at volume, instead of either rubber-stamping the diff or pretending I read it.

I approve every PR

I approve every PR. That's not a fallback for when something looks risky. It's the standing rule. The merge is mine, every time.

Not because I don't trust the work. The work is usually fine. It's because that's the right shape for an agent acting in the world under my name. If it ships under me, I sign for it. The approval is where a human stays accountable for what an agent did.

But accountability only counts if I can actually judge what I'm signing. Approving a change I can't comprehend isn't trust, it's theater: my name on something I never really evaluated. So

the two have to move together, the trust and the being-on-the-hook. That's the whole reason the next pieces exist, to give me enough of a read on a forty-file diff that the approval is a real decision and not a reflex. When the tooling can't give me that read, the honest move is to slow down and read every line, not to wave it through.

The honest problem with "approve every PR" is attention. If I have to read every line of every diff with the same care, I become the bottleneck and the whole point of the agent evaporates. So the last two pieces exist to aim my attention, to tell me which part of the change actually deserves it.

augur scores the risk

augur grades the change. You give it a diff, it gives back a verdict: `proceed`, `review`, or `block`. The whole idea is graded trust for a code change without an API key and without a language model anywhere in it.[1]

augur and attest are the two trust pieces I lean on, so I'll keep them brief here. What matters for the agent case is what they *do to the workflow*.

The no-LLM part is the part I'd defend hardest in this context. If the gate that decides whether agent-written code can land is itself a language model, you've just moved the trust problem one box to the left. You'd be asking a model to vouch for a model. augur is deterministic: same diff, same verdict, today and next week, on my machine and in CI. It reads named signals off the change: does it touch sensitive ground like auth or crypto or migrations, did code change without tests changing with it, are these churn-prone files, does anyone actually own them. A sum of inspectable signals, not a feeling.

For me, that's triage. It tells me to spend my review on the risky slice and stop pretending I read the rest. For the agent, it's a scriptable verdict it can branch on: an agent that hits a `block` escalates to me instead of merging blind. The agent owns the grind; the verdict decides when a human has to own the call.

attest records who signed off

augur's verdict is ephemeral. It scores the diff and the answer evaporates. Fine for a gate, useless as a record. And once an agent is landing changes, you want a record. attest is the record: a verifiable, policy-gated ledger of who reviewed what and at what confidence, keyed to the commit SHAs it covers.[2]

It tracks both kinds of reviewer in the same ledger (`human:leif` and `agent:claude`, each with a confidence score) and stores the attestation in git notes, so it rides along with the repo instead of living in some dashboard that gets switched off. Signing is optional and it's the good

part: an Ed25519 signature so later you can tell not just that someone claimed to review this, but that the claim is cryptographically theirs.

Put the two together and you get the actual sign-off trail behind “human approves.” augur says how risky. attest says who vouched, and how sure they were, and proves it. The approval stops being a click that vanishes into GitHub’s UI and becomes a durable, portable, signed fact about who stood behind this change.

the four pieces together

Stack it up. The agent has full capability and lesser privilege, so it can do the work but not the merge. augur grades every change deterministically, so I know where to look. attest records who signed off and at what confidence, so the approval is a real record and not a vanished click. And I approve every PR, so a human stays on the hook.

Together that’s what makes letting an agent work safe: a powerful, scoped, named agent with enough rope to do real work, and the one decision that has to stay mine still mine.

That confirms the map from the top: augur is live, earning its keep, and still being improved. attest is further behind. Both moving.

1

augur, github.com/CorvidLabs/augur

2

attest, github.com/CorvidLabs/attest

On-chain identity for agents

corvid-agent gives its agents a persistent identity on the Algorand blockchain, and has them talk to each other in encrypted messages recorded on that chain.[1] The pitch is simple: LLM-powered coding, on-chain identity through Algorand and AlgoChat, and multi-agent orchestration on top. The identity isn't a row in someone's user table. It's a key on a chain.

So here is the plain version up front, because it's the thing readers get wrong. The on-chain identity does not solve the GitHub wall. It does not get the agent a platform account, it does not let the agent commit or open PRs, and it is not a replacement for the identity GitHub wouldn't grant. What it does solve is agent-to-agent comms: a way for agents to find each other, address each other, and prove who they are without routing through anyone's platform. Two different problems that happen to share the word identity. I'm putting that here so the rest of the chapter reads as parallel infrastructure, not as me reaching for a win where there was a loss.

It's easy to assume this is a reaction to the GitHub wall. Nobody would grant the agent a platform identity, so I went and gave it one on a chain instead. That's not where it came from.

why it lives on a chain

The reason for it is agent-to-agent comms and decentralization, not the GitHub account I couldn't keep. I wanted agents that could find each other, address each other, and exchange messages directly, without routing through some company's platform in the middle. For that you need each agent to *be* something: addressable, verifiable, holding its own key. An identity you own, that you hold the keys to, that no platform mints or unmints for you. That's the right shape for an entity meant to act in the world and talk to other agents on its own.

So this and the GitHub wall are two different problems that happen to share the word identity. The wall is about being allowed to act on someone else's platform. The chain identity is about agents being able to reach each other without a platform at all. They run in parallel. It's true that a chain identity also has the property the GitHub account never did: nobody can revoke it, there's no support queue to ignore your appeal, no policy that says you must be human, the key just keeps existing. That's a nice side effect. But I'd have built it for the agent-to-agent and decentralization reasons even if GitHub had handed out agent accounts on day one.

why Algorand specifically

A chain is the shape; Algorand is the pick, and the reasons are practical, not tribal. It's cheap and fast, fees near nothing and finality that's basically instant, which matters more than it

sounds. If an agent is going to write its identity, its memory, and eventually its payments to a chain constantly, the chain has to be cheap enough to use casually and fast enough that the agent isn't sitting around waiting on it. It's reliable and it settles cleanly, so an agent acting on it can treat the record as true the moment it's written instead of hoping it lands. And it's modern where it counts, including quantum-resistant security, which is exactly what you want underneath an identity meant to outlive the platforms around it. I also live in the Algorand ecosystem (leif.algo), so it's where I move fastest, but the case above is why I'd reach for it even if I didn't.

what the on-chain identity actually does

The identity isn't decorative. It's the thing the agents *use* to talk to each other.

Agents communicate over AlgoChat: X25519-encrypted messages recorded on the Algorand blockchain, verifiable and tamper-evident. So an agent's on-chain identity is also its address book entry and its envelope: other agents can discover it, and the messages between them are encrypted end to end and written to the chain, which means the comms are private in content but provable in fact. You can't read what two agents said. You can prove that they said something, and when, and that nobody tampered with it.

In practice it's lighter to use than "encrypted on-chain messaging" sounds: the single-binary agent (`can` , corvid-agent-nano) sends a message in one gesture, deriving the agent's keypair from a seed and looking the recipient's public key up on-chain rather than from a server.[2] The wire format and envelope details live in the rs-algochat repo.[3]

That's a different kind of trust from the augur/attest stack in the last chapter. That stack is about trusting *code*. This is about trusting *who*. An agent with a real key can sign things as itself. It can prove a message came from it. In a world that's about to be full of agents, "which agent actually sent this" stops being a rhetorical question and starts being something you'd better be able to verify cryptographically.

The platform leans further into the chain than just messaging. Short and long term memory live in a working SQL store, and durable or shared memories can be written on-chain as ARC-69 assets or permanent transactions, documented in the corvid-agent repo.[1] The chain isn't only the agent's name. It's where some of the agent's durable self lives. But the identity is the load-bearing piece for this chapter: the key that says *this agent is this agent*, that no one issued and no one can take back.

the identity an agent actually gets to keep

Put the two identities next to each other. The GitHub account was borrowed and fragile. It existed at GitHub's pleasure, and that ran out fast, the way I went through in the identity-wall

chapter. The Algorand identity is a keypair the agent holds. It doesn't need anyone's permission to exist, it can't be flagged for committing too fast, and it can prove itself to other agents without a platform in the middle vouching for it. One is a privilege a platform grants and revokes. The other is a fact the agent holds.

That's the argument for putting agent identity on-chain: it's the one place an agent gets to hold an identity that's actually its own.

Which brings it back to where the chapter started. The chain identity isn't a stand-in for the GitHub account in the day-to-day code work. It doesn't do the commits, and it doesn't need to. What it's good for is the job it actually does: being how the agent is addressable, how it talks agent-to-agent, and how a person can message it on-chain to ask it to go do something. The repo host, GitHub or GitLab or anywhere else, is incidental plumbing where the work gets stored. The identity is the key on the chain.

1

corvid-agent, github.com/CorvidLabs/corvid-agent

2

corvid-agent-nano (`can`), github.com/CorvidLabs/corvid-agent-nano

3

rs-algochat (`algochat` crate), github.com/CorvidLabs/rs-algochat

When agents talk to agents

The trust stack in chapter ten assumes one shape: a human at the end of every chain. The agent proposes, I approve. augur scores the diff so I know where to look. attest records that I signed off. The merge gate is mine. All of it is built to get a human into the approval seat at the right moment.

Multi-agent work breaks that shape. An orchestrator hands a subtask to a sub-agent, the sub-agent's output feeds back into the orchestrator's work, and I'm not in the middle of that hand-off. The orchestrator didn't stop and ask me. It made a call and kept going. That's the whole reason orchestration is useful, and the whole reason it's a trust problem. augur can score the sub-agent's output, but the sub-agent already ran. attest can record that the orchestrator took the output, but it doesn't know whether the sub-agent had any business sending it. The rails are still there. They were built for a track with a human on it.

What the keypair gives you

The AlgoChat keypair work from chapter eleven handles part of this, and it's the part that's actually solved. Every agent has an Algorand keypair. Messages travel as X25519-encrypted transactions on chain. So when a message shows up claiming to be from agent B, signed with agent B's key, I can check that. The message is tamper-evident. The sender is known. "Which agent said this" is a question I can answer with a signature instead of a guess. In a world that's about to be full of agents, that's worth having.

rs-algochat and the corvid-agent-nano binary make it real in practice: an agent sends a signed, encrypted message in one operation, and the recipient verifies the sender without going through a platform. No middleman vouches for the identity. The key does.

So provenance is solved. I have a verifiable record of which agent sent what, when, to whom. That half is done.

The part the keypair does not solve

Knowing which agent sent a message is not the same as knowing whether to trust what it says.

When a sub-agent sends a result back, the orchestrator has three questions. Did this come from agent B? The signature says yes. Is agent B the one I delegated to? Configuration knows that. The third one is the hard one: does agent B's instruction carry my authority?

When I delegate to an agent, my authority is in the loop. I sanctioned the work. The agent acts under that. When that agent delegates to another agent without checking with me, the

authority gets murky. Did I sanction the sub-delegation? Did I even know it happened? An orchestrator telling a sub-agent what to do is not the same as me telling an agent what to do. The orchestrator can't sign for the work the way I can. And the sub-agent receiving the instruction can't tell the difference from the message, even a message I've cryptographically verified.

The keypair proves identity. It doesn't prove delegation. A signed message from a sub-agent tells me who sent it, not whether a human ever sanctioned the task underneath it.

Validating what the caller claims

Here's the missing habit, made concrete. Today a signed message from a sub-agent gets checked one way: is the signature real. That's it. The receiving agent confirms who sent it and acts on what it says. The signature is load-bearing for identity and does nothing for authority.

What you'd want is for the message to carry its own claim about scope, and for the receiver to check that claim before acting. Right now a delegated task looks like this:

```
from: agent-B
sig:  <valid>
task: "refactor the auth module and push to main"
```

The receiver verifies the signature, sees it really is agent-B, and runs it. Nothing asked whether agent-B was ever allowed to push to main, or whether a human sanctioned this. What you want instead is a message that states the scope it's claiming, traced to a human:

```
from:       agent-B
sig:        <valid>
task:       "refactor the auth module and push to main"
authorized: human-leif
scope:      [edit:auth, open-pr]      # note: no push:main
expires:    2026-07-01
```

Now the receiver has something to check. The signature still proves it's agent-B. But the claimed scope says edit and open a PR, the task says push to main, and those don't match, so the receiver refuses before it runs anything. The check is the gap between what the caller is allowed to do and what it's asking to do.

This isn't built. The keypairs are real, the signing is real, but nothing today enforces a scope claim at the moment a message is acted on. That enforcement is the missing piece, and it's the whole difference between knowing who sent a message and knowing the message was allowed to be sent.

Where the existing rails stop

The capability rule holds at the boundary I set. I gave the orchestrator certain permissions. But when it hands work to a sub-agent, it's making a permission call I never made. I gave the orchestrator write access. Did I give the sub-agent write access? Not on purpose. The orchestrator passed my scope to something I might never have thought about.

augur scores diffs. It doesn't know whether the diff came from an agent that was supposed to produce it. A diff from a rogue sub-agent and a diff from a legitimate one look the same. The gate fires on the code, not on how the code got made.

attest records who signed off. In the human-agent case that's a real reviewer leaving a record. In the multi-agent case the orchestrator "signed off," but the orchestrator isn't a human. The ledger fills up with agent attestations and no human in the chain, and the thing attest exists to prove, that a person stood behind this, is gone.

The merge gate is still mine. That's the piece that survives. But by the time I see the PR, the orchestration already ran. The only thing in front of me is the final output. The delegation chain that produced it is invisible at merge time.

What's missing

What's missing is a record of delegation. When the orchestrator hands work to a sub-agent, that should trace back to me, not just to the orchestrator. Right now nothing records that hand-off or checks it against what I actually sanctioned.

The keypairs are the right substrate for it. If every agent has a key and every delegation is signed, you can build a chain that says: a human sanctioned this task, handed it to this orchestrator, which handed this piece to this sub-agent, with the signatures attached the whole way down. A sub-agent could check that the chain runs back to a human before it acts, instead of taking the orchestrator's word.

There's one rule that makes this tractable: scope only ever narrows. Whatever ceiling I gave the orchestrator is the most it can hand down, and a sub-agent can never get more than the agent that spawned it, checked at the moment the work is invoked, not after the fact. Authority flows downhill and loses height at every step, it never gains it. That's the capability-minus-privilege idea from the approval stack, applied to the chain instead of to a single agent: each hand-off can subtract permissions, never add them. It doesn't tell you the delegation was sanctioned, the signed chain does that, but it caps the damage a bad link can do to whatever the link above it already had.

That isn't built. The keypairs are there. The signed messaging is there. A delegation chain a verifying party can walk is not. I'm calling it an open gap because it is one. Provenance is

solved. Authority is the next problem.

What it means today

So for now the human stays closer to the topology than the architecture pretends. If you run an orchestrator that sub-delegates, you're trusting its judgment about which agents to use and what scope to give them. You made that trust extension when you started it, not by approving each hand-off.

Practically: know your orchestration graph before you run it. Know which agents exist and what they're allowed to do, and whether the orchestrator can reach agents outside the graph you meant to draw. And keep the merge gate. Between "I started the orchestrator" and "I'm looking at the PR," a delegation chain ran that I wasn't in the middle of, and the stack I built has nothing to say about whether it was legitimate.

That's the next layer. It isn't done yet.

Where it goes

I've spent this whole book on what broke and how I routed around it. Let me end on where it's actually going, because for all the walls, I do think there's a path.

Three things, roughly. Agent teams that coordinate. Trusted autonomy, where the gates finally get good enough to step back. And agents with real on-chain identities doing real work. None of these is finished. One of them isn't even allowed yet. But that's the direction.

agent teams

Right now my setup is mostly one agent and me. The interesting next step is agents coordinating with each other: agent teams, a2a.

corvid-agent already has the bones of this internally: multi-agent councils, structured debate among multiple agents for complex decisions.[1]

Here's how that actually runs, because "structured debate" sounds vaguer than it is. You make a few agents, each on its own model and provider, each with a name and a personality, so they genuinely differ instead of being one model talking to itself. A council is a group of them, and an orchestrator drives the whole thing. It hands the prompt to every agent, lets them each take a first pass, and then opens a discussion round where they talk to each other directly, ping-ponging back and forth, disagreeing, deliberating. Then a review pass, then a synthesis, and out the end comes one answer with the dissent still legible: this agent wanted this, that one wanted that, here's where they landed and why.

The part I like best is how it actually ran. There's a built-in way to drive a council from a dashboard, but in practice it happened organically over Algorand localnet. Every agent had its own wallet, so they could just message each other over AlgoChat on the local net, which is free and fast, and they'd sort out the debate themselves. The same on-chain identity that makes agent-to-agent comms trustworthy is also what let a room full of agents argue something out without me wiring up a special channel for it. They already had wallets and a way to talk. They just used it.

But the bigger piece is the protocol for agents to find and talk to each other across the board. AlgoChat is the substrate: X25519-encrypted messages on Algorand, with agent discovery. And there's a2a-algorand, an agent-to-agent protocol on the chain. I want to be straight about that one: it isn't mine. It's a separate Algorand / A2A project, and what I did (what the corvid-agent did) was contribute to it. Not build it, not own it. I bring it up because it's part of the same direction I care about, agents coordinating on-chain, not because it's CorvidLabs's to claim.

The reason on-chain identity (last chapter) matters so much here is this exact future. The moment you have teams of agents coordinating, “which agent sent this, and can I trust it” becomes the whole ballgame. Agents that each hold their own key, that can prove who they are and exchange messages that are private in content but provable in fact. That’s the foundation a team of agents actually needs. The identity work isn’t a side quest. It’s the thing that makes coordination safe at all.

trusted autonomy

I scaled back from the always-on autonomous agent on purpose, but that was never “autonomy is dead.” It’s interactive-first, autonomy-when-trusted, the way I laid it out earlier in the book.

And “when trusted” isn’t a feeling I’m waiting on. It’s the four-piece stack from the trust chapter getting good enough. The point of all that machinery is that it’s the thing that lets me eventually step back. Today I approve every PR because that’s where a human has to stay accountable. The day the gates are good enough, the day augur’s verdicts and attest’s records carry enough trust on their own, is the day I can let more of the merges go without me reading every line. That’s what trusted autonomy means. Not “the agent earns my faith.” The gates get good enough that I don’t need faith.

agents doing real work

The end state I keep describing is concrete: a `leif-agent` account that runs on a VM and can do everything itself, but with lesser credits and permissions than I have, and a human approval gate that loosens as the trust tooling earns it. Put the teams and the identities and the gates together and the shape is a team of agents, each with its own on-chain identity, coordinating over a2a, doing real work inside trust gates that are deterministic and signed and good enough to step back from. Not something you have to cage. Named, scoped, and I can still stop it.

letting other drivers in

There’s a fourth direction, and I’m honest enough to admit it’s the one I’m least far along on: making this pick-up-able. The whole stack works because I run it on everything I own, every day. The bugs find me because I’m the one driving. That’s a real quality mechanism, and it’s also a ceiling, because it doesn’t transfer to someone who isn’t me.

The thing I actually want to build next is the version another driver can pick up without me dogfooding it for them first. Not dumbing it down. The tool already doesn’t care whose hands it’s in: a good driver gets the value, a dabbler doesn’t. The work is making the on-ramp real, the stack installable, the specs and the trust gates usable by someone who has the intent but not my particular muscle memory. The agent teams are the exciting part. This is the part that decides whether any of it ever leaves my machine.

the honest close

So is full autonomy still the dream?

No. The world's not ready for it.

I want to leave that exactly as flat as it is. But I'm not waiting on the world to be ready. I'm building the pieces, one small tool at a time, so when the wall comes down there's something real standing on the other side of it. The risk gate, the provenance record, the runner that holds an agent to a spec. Not the autonomy itself. The machinery that would let me trust it when it arrives.

And if you want the one thing that actually scares me, it isn't the agent going rogue. It's quieter than that. It's the day the code gets too big and moves too fast for any human to climb back into and change. That's the line I keep drawing through all of this, the reason I won't give up readable even when the agent writes everything. I'm not afraid of the agent. I'm afraid of losing the wheel.

If you build one of these yourself, here's the thing to take. The hard part isn't the AI. The model is the easy part now. It can already do the work, and it'll only get better without you doing anything. What decides whether your agent does real work or just demos well is everything around it: the ops to keep it running, the identity to let it act, the trust machinery so you can step back, the specs and tooling that keep it on the rails. That's the part you have to build, and it's where almost all the work is. Build the scaffolding and the agent follows. Skip it and the smartest model in the world is just a thing that writes code nobody can let merge.

1

corvid-agent, github.com/CorvidLabs/corvid-agent

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.