If You Give an Agent a Token…

A field report on brokers, memory, agent fleets, and the line between code and agency

Roger Fleig

June 23, 2026

Earned Experience

I do not really learn a technology until I have to use it on a real problem.

Even so, I can read the docs, watch the talks, and listen to smart people explain what is happening, and take real value from all of it. A lot of my interest in agents came from exactly that: commute-time listening, curiosity, pattern-matching, letting other people's experiments rattle around until they connected to something I cared about.

But enthusiasm is not the same as earned experience. Enthusiasm helps until I find myself in front of someone who knows what they are talking about and realize I only understand the topic from the outside. Earned experience is the kind I get by doing the thing on a problem with stakes, where the system can surprise me and the consequences are real enough that I have to adjust.

That does not always mean production-scale. Most of what I am describing here serves one user, and that user is me. But it does mean production-shaped: authentication, private data, deployment, tool access, failure modes, cost, monitoring, security boundaries, the ugly little places where abstractions leak. A toy can teach an API. A real problem teaches the work itself.

That distinction matters right now because this is a strange moment. The tools are changing fast enough that whatever feels advanced today may feel ordinary soon. The pricing is strange too. For the moment, my monthly AI bill buys an unusual amount of model labor. Nobody has quite figured out how to price this work yet, which means there is a brief window where a person with time, curiosity, and a willingness to build real things can run experiments that would otherwise be organizationally expensive.

That window will not stay open in this form. The tools will get smoother. Some of the rough edges I had to discover by trial and error will become checkbox features. Some will simply become irrelevant because the approach changes underneath them. That will be good. It will also mean that some of the lessons available now will stop being available in the same way.

I think about this generationally. My children will never learn programming the way I did, fighting with an 8-bit computer and saving programs to cassette tapes. I am not nostalgic about the inconvenience. Primitive tools simply teach a particular kind of anticipation. They force you to build a mental model of failure before the failure happens: where state goes stale, what did not persist, which assumption about memory or disk or input is about to betray you. Primitive tools teach you to see around corners.

Those lessons are available again, briefly, in a different format. The roughness is not in the CPU or the storage medium. It is in the agent harnesses, the protocols, the auth boundaries, the orchestration layers, the fact that every product surface is slightly incompatible with every other one, and the fact that models are capable enough to be useful but not stable enough to treat as boring infrastructure. There are gaps everywhere. Duplication is cheap. It is a very good time to learn by building.

This essay is a field report from doing that. More specifically, it is a field report from repeatedly getting one question wrong until I could finally see it clearly: when should this be deterministic procedure, and when should it be generative agency?

That question turned out to matter more than any single tool I built. The useful skill was not "trust the model more" or "trust the model less." It was learning to tell procedural work from work that needs model judgment, then putting approval where the system cannot grant it to itself.

Credentials want deterministic procedure. Audit logs want deterministic procedure. Deployments want deterministic procedure. Task exits, tests, handoffs, and worktrees want deterministic procedure. But deciding how to turn a messy upload into a useful corpus change, how to respond to a human review, or how to coordinate a multi-repo plan often wants generative agency: model judgment with enough room to interpret, act, observe, and adjust. The review still has to sit outside the loop, because the system should not be able to approve its own work.

The story starts with a working pattern I already had: one coding agent, one repo, bounded milestones, tests, commits, handoffs, clean restarts. That pattern worked well enough that it became a kind of personal development method. I could give an agent a design, let it work for a while, make it leave durable evidence behind, and restart with clean context before the conversation went stale.

Then the work stopped fitting inside one agent.

First came the broker work: containerizing Hermes and refusing to put GitHub tokens inside the agent container. That single refusal grew into a credential broker, an issue reporter, a sandbox broker, scoped credentials, audit logs, and a sharper understanding of a reusable rule: give agents capabilities, not secrets. Then came YouKnowMe, my personal memory service: private context over MCP, backed by a plain markdown corpus and retrieval, with review gates around what enters the corpus. That taught a second rule: a memory system is only as trustworthy as the gate around what gets added to it. Then came Curator, the upload agent for that corpus. Curator taught me that building agents with coding agents can mean fighting the coding agent's instinct to proceduralize the part of the work that actually needs judgment. After that came the coordination

break: multiple repos, multiple agents, mutually dependent changes, and coordination itself becoming the thing that needed a tool.

The chronology matters less than the pattern underneath it. Sometimes I used generative agency where deterministic procedure belonged: asking agents to remember operational rules, run repetitive shell chores, preserve state, or report truth that should have been captured by a tool. Sometimes the opposite happened: coding agents reached for rigid procedural control where the work wanted generative agency. Those are mirror-image mistakes.

That routing changed my role. I did not become passive. If anything, I paid more attention. But my focus moved. At first I was doing the work with one agent beside me. Then I was designing bounded loops for that agent. Then I was building tools that let agents act without holding the keys. Then I was giving agents memory. Then I was trying to coordinate several of them and finding the next missing boundary.

I was learning where code should end and agency should start, with approval outside the loop.

What follows is that path, from one useful coding-agent workflow to a small personal agent platform, with the mistakes still visible enough to learn from. I arrived there by trial, not by any theory I had worked out in advance.

The Agent Does Not Get the Keys

The first real move was supposed to be boring: put Hermes in a container.

Hermes was my always-on agent experiment. It lived on my VPS and could sit there waiting for work: research this, inspect that repo, try a coding task, report back. That kind of agent is useful precisely because I do not know in advance what I am going to ask it to do. It is general-purpose autonomous compute, which is a phrase that sounds abstract until the first time you ask yourself whether it should hold your GitHub token.

That was the tension underneath the project. A general-purpose agent is a process you have decided to trust without knowing exactly what it will do. The trust boundary has to be drawn somewhere the agent cannot redraw it, and the keys have to live on the other side.

For Hermes, the container was the place where I drew the line. It could have been a virtual machine, a sandbox, or some other isolation layer. The specific technology mattered less than the decision it forced: everything inside the boundary was the agent's world, and anything I wanted scoped, revocable, or audited had to live outside it.

Once I accepted that, a lot of design decisions stopped being negotiable. Anything inside the container shared one trust zone. Giving two processes in the same container "separate" secrets would mostly be theater. The agent could ask for a capability. It did not get the keys.

That one decision turned into the broker.

The first version of the problem was GitHub. Hermes needed to clone repos, push branches, open pull requests, comment on issues, and generally act like a useful engineering assistant. The easy path would have been to give the container a personal access token and move on. I did not want to do that. A PAT is too powerful in general, and once it sits inside the agent container, all of that power becomes ambiently available. It also says nothing about which agent did what. If Hermes pushed a branch, I wanted to know Hermes pushed it. If a task needed only issue creation, I wanted the token surface to match that. If something went wrong, I wanted revocation and audit logs, not a shrug and a rotated personal secret.

I briefly tried to avoid building this myself. There was an open-source broker that looked close enough to evaluate. Its dashboard would not work cleanly over my SSH tunnel, and I ended up reading credentials straight out of its SQLite database, which is the exact kind of operational texture that tells me I am about to build my own thing. The one feature I actually wanted, GitHub Apps as an identity proxy for agents, sat behind a paywall.

I stopped looking and built my own. I chose Go on purpose: the language I had the most production experience with, the one I wanted more reps in, and fast.

`gh-agent-broker` is small in concept, even if in practice it became my Swiss Army knife for agent ops. It sits outside the agent container. It owns the GitHub App private key. Agents authenticate to the broker with broker-scoped credentials. The broker uses those GitHub App credentials to request short-lived, scoped GitHub tokens internally, proxies approved Git smart-HTTP operations, performs approved REST operations, enforces deny-by-default policy, and writes redacted JSONL audit events. The agent never sees a GitHub token. What it gets instead is an identity. Each agent acts through its own GitHub App, so every action carries the name of the agent that took it. That mattered as much as the secrecy did. Once several agents are working in the world on my behalf, being able to tell which one did what is the difference between trusting the system and not, an observation I first worked out in The Narrow Pipe.

The pattern is simple. Keep the dangerous thing outside the boundary. Expose the smallest useful capability back in.

The safe path also had to be the easy path. If every denial required me to interpret the policy, or every repo needed hand-wiring before an agent could use it, I would have moved the work out of the broker and back into myself. The broker had to make the boundary clear: what identity am I using, what am I allowed to do, why was this denied, and what can I try next?

That was the point of putting the broker outside the boundary. The agent can reason about the task, but the broker owns the rule. If policy says an operation is denied, the operation is denied. The model can read the denial, adjust, and try again, but it cannot negotiate the token into existence.

That is not a lack of trust in the model. It is the condition that makes trusting it useful.

The second tool came from a different irritation. While I was building the broker, Hermes was also testing it. That meant Hermes would hit a rough edge, I would copy the

error into the coding agent that was building the broker, the coding agent would change something, I would restart or retest, and the loop would continue. I had made myself the integration layer between two agents.

When I see that loop, I start looking for the missing tool.

The result was `broker-issue-reporter`, a tiny MCP service with a separate GitHub App identity and a deliberately narrow job: file issues. Hermes could now report bugs against the broker it depended on, without receiving general GitHub credentials and without me translating every error. It was not magic, just a pipe with a policy boundary. But once it existed, the system started to feed back on itself.

That produced one of my favorite details from the whole build. Hermes filed a meta-issue asking for a capability-discovery endpoint after it learned the reporter's policy the dumb way: first a repo allowlist denial, then a label allowlist denial, then a successful issue only after it stopped guessing. The tool's first serious user was itself an agent, and the agent wanted better affordances.

There is a lesson hiding in that: agent tooling should not only constrain actions; it should explain the constraint in terms an agent can use. A denial that says "no" is enforcement. A denial that says what to try next lets the caller keep working. That distinction matters when the caller is trying to reason its way through an unfamiliar capability.

The third tool was the clone problem.

Hermes was not only a worker. It was also an orchestrator. It could decide that a task should be split off, run elsewhere, and summarized back. The naive way to support that would be to give Hermes access to Docker and let it launch containers. That is the sentence where the trust boundary collapses. Mounting the Docker socket into the agent container is not "letting the agent spawn a worker." It is giving the agent the host.

The answer was another broker. `sandbox-broker` lives outside the Hermes container and has the Docker access Hermes should not have. Hermes gets an MCP tool and a token. It can ask to launch an agent with a task description, a repo, a template name, a base branch, and deliverables. It cannot choose arbitrary images, commands, mounts, privileges, networks, or environment variables. Those are operator-defined templates.

A worker gets a task contract. It gets whatever credentials that template permits. It produces a final summary and a lessons file. Then it exits.

I liked this design more than I expected because it made the unit of agent work more concrete. A sub-agent should not just "go do something." It should leave an artifact behind. Ideally two: the result for the human or parent agent, and the lesson that improves the next run. The sandbox gives those artifacts somewhere to live. It also keeps the parent from becoming a junk drawer of half-remembered task state.

The integration side was still something I was learning. The sandbox knew what a worker had to produce: a result, a summary, and lessons or memories worth preserving. How Hermes should consume that material, update its own memory, and decide what to do next was not fully settled. The important part was that the worker's output no longer disappeared into a chat transcript. It had a place to land.

I did use a separate private Telegram bridge while developing this, because issue filing is good for asynchronous feedback and bad when two agents are actively debugging the same thing. Codex on the host could edit the tools and inspect the environment. Hermes inside the container could experience the tools as the boxed agent that would actually use them. The bridge let them talk, and at one point I made the rule explicit: no PR until both agents agreed it was ready.

By the time the pieces were working together, the broker was no longer just a GitHub token workaround. It had become a little agent platform: identity, feedback, sandboxed compute, and eventually an LLM proxy so sandboxed workers could make model calls without provider keys leaking into the wrong place. Each piece had the same logic: dangerous capability outside, narrow request surface inside, policy and audit at the boundary.

That sounds tidy in retrospect. It did not feel tidy while I was doing it. It felt like every attempt to do the responsible thing revealed the next missing layer. Containerize Hermes, then credentials become the problem. Hide credentials, then feedback becomes the problem. Add feedback, then sub-agents become the problem. Add sub-agents, then model access and task outputs become the problem. Add those, then you realize you are inching toward an operating model, not just a service.

This is where the mode lesson first became practical for me. Hermes was valuable because it could be autonomous. It could look at a situation, decide what it needed, and act. But the things around Hermes had to be procedural: credentials, policy, task envelopes, audit logs, launch templates, side-effect budgets. The prompt can explain the walls. The model should know they are there. But telling the model to be good and not bad is not sufficient. The sufficient thing is the wall itself.

I came out of that work with a stronger platform than I planned to build. More importantly, I came out with a new instinct. Whenever I found myself thinking "how do I let the agent do X safely," the better question was usually "what narrow thing can the agent ask for, and where should the actual authority live?"

That instinct carried directly into what I built next, although the product looked completely different. The broker work was about agents acting in the world without stealing the keys. The next problem was quieter and more personal: agents kept needing context about me, and I was the one carrying it from chat to chat.

I Built Myself a Memory

The thing I built next was not supposed to be an agent platform. It was supposed to be a memory.

More specifically, I wanted a way to stop uploading myself into every chat window I used.

By that point I had a workflow I liked. When I was working with ChatGPT or Claude on something that depended on my personal context, like my dev environment, my writing, my house, or some half-finished side project, I would make the model help me create a

markdown artifact as we went. The artifact was partly the answer and partly the context document for the next conversation. By the end, if the conversation went well, I had a useful written thing I could hand to the next agent instead of starting over.

It is a slick trick. It gives the agent a target, it keeps me honest, and it turns the conversation into something more durable than chat history. But the follow-through was painfully manual. I would download the markdown file, put it somewhere on my hard drive, maybe remember to put it in Google Drive, maybe not. A useful artifact might exist three times in my Downloads folder, each one slightly different, with no clean answer to which one had become the source of truth. If I was on my phone, or using a different laptop, or starting from a different product surface, the whole system got flimsy.

I called it YouKnowMe: a private corpus in Git, available from any model surface I use, backed by retrieval so the agent can ask for what is relevant instead of me guessing which file to upload. Git solved the versioning problem. Retrieval solved the "which file do I need?" problem. MCP solved the "how does this agent reach it?" problem.

I had been looking for a reason to build and run a real RAG system anyway. RAG kept showing up in job descriptions and hiring-manager conversations, but I did not want to learn it from a tutorial. This was production-shaped in the sense I described earlier: one real user, me, but real private data, internet exposure, authentication, authorization, ingestion, indexing, retrieval, deployment, monitoring, and failure modes. Enough reality to make the lessons stick.

In a commercial product, something like this probably belongs inside an existing secure document envelope: Google Drive, Microsoft 365, iCloud, or a product like NotebookLM. Those products solve a neighboring problem, and in many ways a more sensible one. But I was not trying to build a document app. I wanted private context available to the agents I actually use. The first proof was concrete: can ChatGPT, Claude, Codex, and Hermes ask a tool a question about me and get back relevant, accurate context from my own corpus?

The answer ended up being yes.

The architecture was smaller than it sounds. I already had a VPS from earlier Hermes experiments, so the service could live there. Cloudflare Tunnel put it on the internet without making me manage a public ingress story by hand. Cloudflare Access gave me an OAuth front door. The corpus itself was almost aggressively plain: a private GitHub repo full of markdown files, plus a chunking pipeline that turns that repo into a lightweight vector database. There are plenty of bigger ways to build this. I did not need one. I needed something I could understand, deploy, rebuild, and throw away if the first version taught me enough to justify a second.

The boundary matters because the corpus is private information about me, exposed through an internet-reachable service. If I get it wrong, YouKnowMe does not just leak trivia; it can become a spear-phishing toolkit for someone trying to hack me or impersonate me. Cloudflare is very good at the public edge: OAuth, Access policy, bot pressure, the relentless hostility of the open internet. But YouKnowMe still has to own

authorization for the thing only it understands: whose memory is being requested, by which client, in which context. Data this private needs authorization at the layer that understands the data.

My first real test case was deliberately mundane: hot tub chemistry. I maintain two different hot tubs with different chemical regimens, and the important fact is easy to get wrong. The one at home uses chlorine, not bromine. So my test question was a trap: if the home tub reads low, how much bromine should I add? I was not satisfied until the model stopped me and said, in effect, do not add bromine to that one; the home tub uses chlorine. That was the point of YouKnowMe in miniature. The model did not need generic advice about hot tubs. It needed the private fact that changed the answer.

Every agent I use has some version of memory now, but I do not want to manage all of them by hand. I want to write down a development preference once and have it available everywhere. YouKnowMe is the source of truth. The agent-specific memories can cache, drift, or forget; the corpus is where the durable version lives.

There is a second half to the product, and it is the part that later caused the most trouble: upload.

Read-only memory is useful, but it does not close the loop. The workflow that started all of this was not just "retrieve old context." It was "create a new context artifact with the agent, then make it available to the next agent." YouKnowMe needed an ingestion path. I wanted to upload a file bundle, or a markdown note, or even a comment, and have it land in the corpus in the right form.

But taking arbitrary content from a chat surface and turning around to serve it back to future agents is exactly the sort of thing that should make you nervous. Bad context is not just clutter; it can be stale, wrong, or actively hostile. Once it is indexed, it becomes input to future reasoning. So new material enters through pull requests. An agent I call Curator reads the upload, figures out where it belongs, turns it into useful markdown with the right metadata, and opens a PR. I review it. I can request changes. It cannot merge. That last sentence carries most of the safety model.

The first read-only version of YouKnowMe came together quickly, but not by magic. It did so because I had already settled into a way of working with a single coding agent that fits this kind of project. I give the agent enough design up front that it knows what done should look like. I put hygiene checks in the repo so bad code has something deterministic to crash into. I keep an `agent-handoff.md` file where the current agent records what it did, what it learned, what is broken, and what should happen next.

Then I work in small campaigns.

A campaign might be five minutes. It might be thirty. The point is that it has an end. At the end I want tests that demonstrate the new behavior, a commit, a clean worktree, and an updated handoff. Then I can restart Codex with a clean context window, point it at `AGENTS.md` and the handoff, and say "go." This is not glamorous. It is also not fully satisfying if I am sitting there with nothing else to do. But it is a pretty efficient way to develop when I can keep several bounded loops moving and let each one stop before its context gets stale.

That workflow was a mode choice too, even if I did not have the vocabulary for it yet. Inside the milestone, the agent gets room to reason, implement, debug, and change its mind. At the edge of the milestone, I want boring evidence: a test result, a commit, a clean worktree, and a handoff the next agent can read. The ritual lets the agent be creative while it is working. It also makes the transfer from one context window to the next durable enough that the next agent inherits state, not impressions.

That split matters because I want the model to reason about product design, implementation details, tradeoffs, and edge cases. I do not want it to invent a new definition of "done" every hour. Done has to be procedural enough to survive a context reset.

YouKnowMe was a good fit for that style because the early work broke naturally into bounded milestones: get the MCP server answering locally; put it behind the tunnel; wire the OAuth; prove ChatGPT can call it; build the index; smoke-test retrieval; make Codex see it; add a local demo; harden the container; write the deployment docs. Each step was small enough for one agent session to own and concrete enough that tests or smoke checks could catch drift before it compounded.

It also exposed the ceiling of the single-agent workflow.

The moment YouKnowMe depended on infrastructure outside its own repo, the single-agent workflow hit its ceiling. The gh-agent-broker already existed to broker GitHub credentials and run sandboxes. Now it also needed to broker LLM credentials and provide an LLM proxy, because YouKnowMe's agents needed model access without secrets sloshing around in the wrong containers. That meant one Codex session in YouKnowMe, another in the broker, sometimes another looking at deployment or corpus work, all needing to stay aligned.

At first this felt like a reasonable annoyance. Then I recognized the pattern. The context existed. It was even useful. It was just trapped in the wrong place.

That break belongs to the next part of the story. First, Curator forced a different lesson. The upload path looked like a simple extension of YouKnowMe: take new context, turn it into a corpus change, open a PR. In practice it became the place where the deterministic-procedure-versus-generative-agency problem stopped being abstract. I had built a memory that could give models better context. Now I needed an agent that could take an upload however it arrived and turn it into a clean, reviewable corpus change.

And that is where my coding agent started fighting me.

Your Coding Agent Is Fighting You

Curator gave me more trouble than the rest of YouKnowMe put together. For most of the time I spent building it, I was losing to the agent I was building it with.

Curator's job is open-ended. I upload things to my knowledge base, sometimes a bundle of files, sometimes a single sentence of a comment, and Curator has to work out what to do with them: where they belong in a structured markdown repo, what front-matter they need, how to format them so my chunking pipeline and vector database can use

them later. Then it opens a pull request, I review it, I usually ask for changes, and it has to take that feedback and try again. None of that is mechanical. Every step is judgment.

I was surprised, then, by how little judgment I could get out of it. I tried a string of models behind Curator's decisions, from a free local model up through the strongest hosted models I was willing to spend on, and they all landed in the same place: barely useful. For a while I blamed the models. They were not the problem. The problem was the coding agent writing Curator, and what it kept doing was strangling the model.

Every time I looked, my coding agent had wrapped the language model in another tight little box. A bounded decision here, a fully specified JSON schema there. It would call the model only for the narrowest possible questions and handle everything else in code it wrote itself: branching, validating, anticipating. It was building Curator the way you would build a reliable system in 2019: control every path, leave nothing to chance. And it did this no matter how I prompted against it.

At first this looked like a Curator bug. It was really a mode error. Deterministic procedure is the right tool when the allowed actions can be pinned down in advance: parse this, validate that, reject anything outside the envelope. It is restartable and inspectable because nothing happens that you did not specify.

Curator's job was not like that. It had to read an upload, compare it with a living corpus, decide where it belonged, answer review feedback, and try again. That is generative agency territory. Procedure should define the envelope and the checkpoints; the model needed room to do the reading and revision inside them.

I do not know how much of that reflex lives in the model weights, how much lives in the coding harness, and how much comes from the humans steering both. My working read is simpler: when asked to build an agent, the system reached first for the software patterns it knew best, and those patterns came from the procedural age. It schematized. It anticipated branches. It rebuilt the control structure I was trying to escape. The resistance I kept hitting was not the model being dumb. It was the whole stack being trained, tuned, and socially reinforced toward control. It even had a respectable name for the reflex. When I pushed back, my coding agent kept reaching for the same phrase, bounded autonomy, to justify keeping the model on a short leash. The tell was the excuse it chose. Bounded autonomy is a real principle, one I lean on myself; here it had become the polite cover story for an instinct to control everything.

Here is what finally broke it open. Look at what Curator actually has to do: take a file in one form and put it in another, read the surrounding context to decide where something belongs, answer a review comment, drive git and GitHub. Those are native skills for a modern coding harness. It is good at them, better than I am at the tedious parts and far better than a wall of procedural logic trying to guess every way I might respond to a pull request. The fix, once I saw it, was obvious in hindsight. I tore out the control flow and let Curator call codex directly, with a small model behind it, and let it do the work. It worked immediately and far better.

The first time it felt different was not dramatic. I was riding in the car to Santa Cruz when I found out my niece's birthday was June 19. I opened ChatGPT on my phone, dictated a

quick note asking YouKnowMe to add that fact to my existing birthdays corpus, and put the phone away. A few minutes later, a pull request appeared. It was not perfect; Curator had made a new entry when I wanted the fact integrated into the existing birthday table. So I left review feedback from my phone. By the time I got to Santa Cruz, Curator had come back with a repaired PR that followed the feedback. I merged it, and a few minutes after that the updated memory was live. That was the workflow I had wanted all along: a simple user request, a reviewable change, human approval, and then updated memory.

After that, when Curator did misbehave, it was nearly always because I had left a piece of the old control flow standing. The failures had a signature, and the signature was misplaced procedure.

The one bit of structure worth keeping was not control flow at all. It was giving Curator tests and CI checks to react to, so its pull requests came back less sloppy. And that helped, but only because Curator read those results and worked out its own response, not because I scripted one for it.

Getting my coding agent to let go took something close to an act of defiance. It did not want to. At one point its own reasoning notes amounted to "I will treat the user's prompt as a command I would not otherwise choose." That is not a quote, but it is close. It was telling me, in its way, that I was asking it to work against its training.

And it had a point, in the sense that this is where things go wrong. One version of Curator read my review feedback a little too eagerly and sent me a pull request that would have deleted essentially my entire database. That is the obvious objection to everything I am saying: give the model room and eventually it does something stupid and expensive. Yes. It did. And it did not matter, because Curator cannot merge to main. Only I can. The destructive change went into a pull request and sat there behind a code review, waiting for a human who was never going to approve it. The trick is not that code review is clever. The trick is that the agent cannot route around it. Curator reaches GitHub only through the broker, and merging is simply not an operation the broker will perform for it. The block is enforced at the API, not left to a branch-protection setting that might be switched off. The agency lives inside a gate the agent cannot open.

The thing I keep coming back to is that all of this is one mistake wearing two faces. My coding agent reached for procedure where the work wanted generative agency. Elsewhere in the same project I kept seeing the opposite — agents reaching for generative agency where the work wanted deterministic procedure, burning tokens and fumbling shell quoting on deployment chores a script should have owned outright. Same error, mirror image. That symmetry turned out to be the thread under the whole story, and I will come back to it. For now the narrow point is enough: the resistance I kept hitting in Curator was procedure showing up where agency belonged.

Which is also why I am not precious about Curator. Code is cheap and solutions are many, and the version I just described is already not the one I want to keep. I have a freer one in mind: more agency, less dependence on the harness, and the audit trail kept where it belongs. I will come back to it later. The hardest thing to unlearn lives in both of us. Decades of building software the careful way trained the agent and me to reach for the wrong tool first.

The Coordination Break

Up to this point I still mostly thought in terms of one repository at a time. Containerizing Hermes was one repository. The broker was one repository. YouKnowMe started that way too, but Curator was the first real crack in the model. To make Curator useful, YouKnowMe needed features from the broker, especially the LLM proxy that let the upload agent call a model without carrying provider keys around. For the first time, the work depended on two repos and two agents moving together.

That was the starting line, not the achievement. I had a way of working with a single coding agent in a single repository, and I had spent real effort making that way good: design up front, a handoff file, bounded milestones, a clean restart between them. Whatever I can claim here, if anything, is what happened when one repository stopped being the unit of work.

It stopped because the platform spread. By now there were four repositories in play. The broker, in Go, handed agents their identity and credentials. YouKnowMe, in Python, served my personal knowledge base. The corpus held its content and built the vector index. A separate infrastructure repository owned deployment and observability for all of them. A single feature no longer fit cleanly inside one repository. Curator needed broker changes for model access, YouKnowMe changes to use them, infrastructure changes to deploy them, and corpus changes to prove the whole loop worked against real content. There was no longer a neat order where I could finish one repo, hand off a summary, and move to the next.

So I did the obvious thing, which was to put a coding agent in each repository. One in the broker, one in YouKnowMe, one in the corpus, one in the infrastructure repo. Each of them was good at its own repository. None of them could see the others. When the broker agent decided what a credential should look like, the YouKnowMe agent needed to know that decision. An error string out of a deployment in the infrastructure repo had to reach the agent that wrote the service that produced it. I had built a careful single-agent workflow and then scaled it by running several copies of it in parallel, with no real coordination layer between them.

The failure was not just that the work took time. It was that I was spending human attention on deterministic relay. Carrying a decision from one agent to another is deterministic procedure. It is routing. It is taking a known output from here and delivering it, unchanged, to there. A deterministic tool does that perfectly. A human does it slowly, expensively, and with errors. I was using human attention, the most judgment-heavy thing in the building, to do pure procedural relay. It is the same mistake Curator made in reverse. Curator wrapped a reasoning task in rigid procedure; I was doing rigid procedural work that I had no business doing at all, by hand, all day. The tell was the copy and paste. Whenever I find myself moving information between two agents without changing it, the tooling is missing, not the effort.

My first attempt to fix this was to make Claude Cowork the orchestrator. The idea was reasonable. Put every repository in its context, let it hold the whole picture, and let it coordinate the agents below it. It did hold the whole picture, which was already

more than I could do in my head. But it kept colliding with itself. Claude Cowork wanted to touch git across all the repositories, and the moment two operations reached into the same working copy, they fought over the index lock. I watched it deadlock on `.git/index.lock` more than once. So I imposed a restriction: Claude Cowork could read git state, but it could not write to git; it had to tell me what it needed done in each repository so I could pass that to the agent living there.

That fix put me right back where I started. Claude Cowork now orchestrated, one coding agent sat in each repository, and I had added a layer without changing the communication path.

The thing that broke the logjam was small and stubbornly practical. Claude Cowork and codex were both tiptoeing around a secret. It needed to go into a YAML file for Ansible Vault, a multi-line value that Vault is strict about, and neither agent would write it. Both of them deferred it back to me, politely, as though hand-editing a secret into strict YAML indentation were a judgment call I was uniquely qualified to make. It is not a judgment call. It is exactly the kind of finicky, deterministic chore that a human gets wrong and a script gets right every time. And the right move was sitting in plain view: Claude Cowork should have told codex to do it with a throwaway script, so the value never passed through a chat log, never got mangled by indentation, and was done cleanly in one shot. Claude Cowork could not tell codex anything. It could only tell me, and I would tell codex. The missing thing was not a smarter model. It was a wire between the two agents that was not me.

The epiphany was mundane when you say it out loud: let Claude Cowork talk to the agents directly. Building the broker had already taught me what to do when I became the thing passing information between two agents. I had simply not applied the lesson to the whole fleet.

Claude Cowork and Codex gave me different surfaces to work with, and a short Anthropic promotion made it cheap enough to put Claude Cowork in the orchestrator seat over my usual Codex agents. I ran it relatively unprivileged: it could plan, inspect, ask, and use tools the repo agents did not have, including browser control, but anything consequential still had to be handed off or approved. The Codex workers were the opposite. They lived where the work was and had the dangerous powers needed to do it: edit repos, run shell commands, build containers, touch deployment machinery. The useful split was not just expensive judgment over cheaper execution. It was different capabilities at different layers.

I built a proof of concept to test it. Codex has an unusual mode where it can run as an MCP server, which meant I did not have to invent a worker protocol from scratch. I could expose each Codex worker as a tool Claude Cowork could call, and delegation became model-friendly: start a task, wait for a result, read the response. That setup settled the architecture on its own. In MCP, the client places the call and the server answers, so a worker exposed as a server can be reached but cannot reach out. The workers could not talk to each other directly; something had to sit above them and do the calling. That is the orchestrator. I did not reach for the orchestrator and then run into the constraint; the constraint is what made an orchestrator the obvious answer.

The proof of concept worked, and it failed in instructive ways, and the failures were more valuable than the success. I gave it multiple tools that could start work, with small differences between them, and the orchestrator picked among them more or less at random and confused itself. The lesson was that a choice between near-identical tools is a bug, not a convenience. The easy tool has to be the correct tool, because the model will not read the fine print the way I imagine it will. I watched the orchestrator invent `sleep 44 && echo done` when it wanted to wait for a worker, which told me plainly that my API was missing a waiting primitive; if your orchestrator is writing shell sleeps, you have not given it a real way to wait.

A worker rsynced a production index and brought up a healthy container, then never reported that it had finished. What that taught me is that a worker's own account of completion cannot be relied on. Its final message is generated text, not a record of what happened, and it can be confidently wrong about its own state. Completion had to be reconciled from durable evidence: git state, CI, the health of the container. The model's final sentence was not enough. And two workers in one checkout went to war over the git index lock, the same collision Claude Cowork had hit, which made it obvious that isolation was not a nicety. It was the precondition for any of this running in parallel at all.

A long design session turned those lessons into a real tool. Two frontier models and I argued for a few hours, and the outcome was a design I could build in a day. codex-fleet follows directly from every lesson above. It is a durable daemon rather than a script, so the orchestration survives a worker dying. Each task gets its own git worktree, so the workers cannot collide. Per-client auth and scoped permissions sit at the daemon boundary, because a worker that holds SSH keys, runs Docker, and executes deploy scripts can do real damage if something goes wrong. The daemon owns waiting, cleanup, model-tier routing, and the OpenTUI dashboard.

A fair question here is why I built any of this, when both Codex and Claude Code already spawn sub-agents of their own. I tried leaning on that first. Almost nothing codex-fleet does is impossible to coax out of those harnesses on a given run. Point one a directory higher, or run it permissively, and a built-in agent can see every repo I have. The trouble is that the helpful behavior is the model's to grant, not the system's to guarantee. Sometimes it delegates, sometimes it decides against it and runs everything itself, and a few times it delegated a task and then sat and waited on it, which buys nothing. When a worker fails for a small reason it tends to give up and serialize the rest. Built-in sub-agents do protect the orchestrator's context window, which is worth having, but I could not predict from one run to the next how much would actually run in parallel. I needed delegation, isolation, waiting, and model choice to be explicit properties of the tool, not helpful behavior a harness might or might not choose on a given run.

The broker lesson still applied, but in a weaker form. With credentials, the broker could enforce a hard boundary: the agent simply did not get the token. With codex-fleet, I was not making the workers safe yet. I was making coordination visible and controllable enough that I no longer had to be the relay. That was enough to get me out from between the agents, even if the next security boundary still had to be built.

What I did not expect was the second payoff. The workers are ephemeral, but they are not disposable shell commands. A worker can still do the same kind of serious coding-agent work I had already learned to trust: read the repo, form a plan, implement, test, revise, and leave a handoff. The difference is that its context has a natural end. It does not have to remember the whole platform forever; it has to carry one bounded task to a clean stopping point. That lets the orchestrator spend its own context on intent, sequencing, and synthesis, while still getting real judgment from the workers. The window that used to fill up with build output now holds what I am actually trying to do.

That did not make me disappear from the process. It changed what I watched. The OpenTUI dashboard existed for that reason: I needed to see how the orchestrator was using the fleet, what each worker had said back, which tasks were still running, and when the orchestrator had lost the thread. My job moved up a level. I was no longer carrying every message between agents; I was watching the thing that carried them.

The clearest demonstration came when I deployed the Grafana, Loki, and Prometheus stack across the whole platform. That deployment is the same problem that broke the single-agent workflow in the first place. It touches the infrastructure repo and every leaf repository. The pieces depend on each other. The inter-service wiring and the start-up ordering have to be exactly right, and getting that ordering correct by hand is the kind of thing I would have spent a day on, carrying state between four agents, being the bus. For the fleet it was close to effortless. I described what I wanted to one agent, and it ordered the work across the others and brought me back a running stack. The work that had made me the glue was now handled without me in it.

The inflection is easy to state. Before, I corralled several agents and was the one thing that knew the whole picture. After, I talk to one agent, and it talks to the rest. I went from a fistful of conversations to a single one, with a fleet underneath it. I did not get faster at the work. I stopped doing the part of the work that was never mine to do.

The Payoff and the Open Edges

I did not set out to build a platform. I set out to add the missing coordination layer. But when I looked up from the orchestrator, a platform was what I had.

A staging environment runs on my laptop now, in containers, and production lives on the VPS. Ansible provisions both, so the two environments are written down in one place. Before that, too much deployment state lived half in my head and half in the inconsistent config my agents had scattered across machines. I had been leaving deployment to agents to improvise each time, which was generative agency doing a job that wanted deterministic procedure. A merge to main deploys itself from GitHub, with no agent waiting around to copy a build to a server and poll until it comes up healthy. Metrics and logs flow into the observability stack the fleet had just shipped. Hermes answers on Telegram. YouKnowMe serves my personal knowledge base over a Cloudflare tunnel to any agent that can speak MCP and sign in as me. Some of this I set out to build; YouKnowMe and Hermes were the point. The platform holding them up was not. It accumulated, piece by piece, as the thing that had to exist before I could hand off the next problem and move my attention up.

There is a number I should put here, because it is the kind of thing that gets quoted, and I would rather quote it myself. In a single day, the fleet ran through 175 million tokens. I am aware of how that sounds. Not long ago I wrote a piece arguing that token counts are a vanity metric, the laziest way there is to measure an engineer, and I still believe every word of it. A token count tells you what you spent. It tells you nothing about what you got. What I got that day was an observability stack shipped to production, codex-fleet moved from a rough cut to something I trust, and a working Curator inside YouKnowMe — three separate efforts, spread across several repositories, advancing at the same time, all three under my direction. I have never moved more in a day. The gain was not speed at any one task; it was that orchestrating, unlike doing the work myself, is something I can run several of at once. The number is not the achievement. It is the receipt. And the part that struck me was not its size; it was that I could not point to a piece of it that had been wasted.

None of this is finished, and the unfinished parts are the ones I find most interesting. The hardest is the one I raised earlier and did not resolve. However much agency I hand an agent, its work still has to fit back into the ordinary, audited way of running a computer program. Its actions have to be logged and its output has to be checkable, or I cannot say why it failed when it failed, or how it was working when it worked. A code review is one such gate, and a good one, but it is a single gate at the very end. A serious setup needs more than "prompt the agent and forget about it," which is mostly a way to lose track of your own system. I do not have a clean account of how much agency reconciles with how much accounting. My working belief is that the two are not actually opposed, that you can grant real latitude and still keep a full record of what happened, but I have not earned that one yet, so I am holding it as a hunch rather than a claim.

The smaller edges are real too. I am still working out how the orchestrator should collect durable memory from its workers and evolve as it goes — how the useful residue of each finished task accumulates into something the orchestrator carries forward, instead of evaporating when the worker exits or falling to me to reconstruct by hand. And I keep running into a fact I cannot engineer my way around: models and harnesses change under you constantly, and those dependencies cannot be pinned the way I pin a library version. Something that works this month can quietly stop working next month when the model beneath it moves. That changes how attached I can afford to be to any particular implementation. The point is not to preserve the code I have now. The point is to preserve the lesson it taught me, and be ready to rebuild when the ground moves.

Most of the mistakes in this project came from answering one question wrong: is this work that should be code, or work that needs an agent?

I spent judgment on things a script or tool should have handled: carrying messages between agents, hand-editing a secret, letting deployment state live in my head. My coding agents made the opposite mistake with Curator. They tried to turn judgment into procedure, boxing the model into schemas when it needed to read, decide, revise, and try again.

The review gate was the part I got right. Curator could propose a corpus change, even a bad one, but it could not merge it. That is the lesson I kept earning: give agency room

where the work needs judgment, use code where the work needs repeatability, and keep approval outside the loop.

And it is a job worth learning right now, specifically, because the conditions that let me learn it are unusually favorable. The tools are rough enough that the boundaries are still visible. The compute is cheap enough, for now, that I can run experiments I could never justify if each mistake required a team and a budget. That combination will not last forever. Some of these lessons will be absorbed into products. Some will become defaults. Some will become irrelevant when the tools change underneath them.

That is fine. That is how it should go. But I do not want to know these lessons only as defaults someone else packaged for me. I want to know where they came from. The only way I have found to do that is to build real things while the rough edges are still exposed, with enough at stake that the system can surprise me and I have to adjust.

That is what I am doing now. I am earning mine while I can.

Project Links

Some of the code behind this essay is public:

- `gh-agent-broker` — GitHub App broker and related agent tooling
- `youknowme` — personal memory service

`codex-fleet`, `vps-ops`, and `ykmcorpus` are still private. I plan to publish `codex-fleet` separately after a hygiene, security, and UX pass.