PORTFOLIO · JULY 2026

Spark — A Body of Work

Everything I build, at home and at work — and the way one person ships it all, with AI as the method.

34
LONG-RUNNING SERVICES

20
DOCKER CONTAINERS

25
SCHEDULED TIMERS

19
PUBLIC SUBDOMAINS

26
GIT REPOSITORIES

0
INBOUND PORTS OPEN

The home platform, sampled live from the box — one slice of the work

Cory Dickes

Introduction

This is a body of work. Not a résumé, not a tidy project list — the actual things I've built, at home and at work, and the way I build them.

The center of it is a home server I call the Spark. One box that's turned into something closer to a small company's infrastructure than a hobbyist's NAS: my media, my search, my DNS, a Windows VM, a handful of real web products, and a growing family of AI systems that watch the box, learn about my life, and help me build the next thing. Everything public runs behind Cloudflare with **no inbound ports open** on my network, fronted by one reverse proxy, pulled together under a single dashboard I can drive from my phone.

But the Spark is just where it's easiest to *see*. The same hands build the tools I run professionally — the automations, integrations, and internal systems that keep my day job in IT moving — and I don't draw a line between the two. It's one craft. So this document doesn't draw one either.

Here's the part that actually matters: I didn't build any of it the way one person normally builds this much. I direct **fleets of AI coding agents** — often several at once, coordinating live on a shared board — and I've built the plumbing that lets them work on the same code without stepping on each other. That's the real story. The projects are the evidence for it.

One more thing. This isn't a sales deck. Where something's a prototype, I say so. Where a design choice got forced on me by a constraint — one GPU, a shared memory pool, a budget of basically zero — I tell you the constraint, because the constraint is usually where the actual engineering lives.

AI Is a Tool for Engineers — Not a Replacement for Them

Let me say the quiet part first, because the whole rest of this document depends on it: **AI didn't replace me. It removed the ceiling on what I can build.**

Everything in these pages was made by one person. A 33,000-line firewall-monitoring platform across 91 modules, watching 100-plus firewalls for 50-some client networks. A disaster-recovery harness that rebuilds an entire server fleet from one file and a passphrase. A dozen vendor integrations that all stayed running. A home platform of thirty-some services and a family of AI agents that heal themselves at 3am. Four years ago, one person shipping that much would've been a joke. Now it's a Tuesday. That's the productivity story, and I'm not going to be modest about it — the gap between what I ship now and what I shipped before AI is not 20% or 50%. It's the difference between "I couldn't take that on" and "done by Thursday."

But here's the part the hype gets exactly backwards. That leverage doesn't make the engineer *less* important. It makes them the whole game.

AI will happily write you 33,000 lines. It will not tell you the polling loop is about to kick your own techs out of the firewall console mid-change. It won't know that acting on "71 firewalls are broken" would take down production at client sites — when the real number was 16 and the other 55 were a bug in my own query. It won't decide that an LLM gets to *propose* a fix but a hard-coded Python whitelist is the only thing allowed to *execute* it. Every one of those calls is judgment, and judgment is the scarce thing. The model is an engine. Someone still has to know where the car should go, and notice when it's about to drive off a cliff.

So what actually changed in the job? The typing compressed and the judgment expanded. I spend almost no time on the mechanical part now — I direct, I review, I verify, I decide what's worth building and what guardrails keep it safe. That's not a lesser version of engineering. That's the *good* part of engineering, the part that was always where the skill lived, finally freed from the eight hours of boilerplate that used to bury it. AI doesn't hide a mediocre engineer; it exposes them, because when everyone can generate the code, the only thing left that differentiates you is taste, rigor, and knowing what "correct" even means.

That's also why I keep the honest caveats in every write-up here — the false positive I almost shipped, the recovery harness that restores infrastructure but not data, the monitoring pipeline that's only as good as its signal-to-noise. A tool that makes you fast also makes you fast at being wrong. The engineer is what stands between the two.

So no — I don't think AI is coming for good engineers. I think it's the loudest possible amplifier for them. Point it at someone who knows what they're doing and you get one person with the output of a team. Point it at a business that treats it as a headcount-reducer instead of a force-multiplier, and you get a very efficient way to ship the wrong thing. I know which one I am. This document is the proof.

Contents

Three movements, one body of work — the home lab, the professional software, and the four years of operations underneath both.

Movement I · The Home Platform

The Spark — a self-hosted home server I turned into a working lab for how one person builds and operates real systems with AI. Everything in this movement is running right now.

The Platform at a Glance

Everything on the Spark sits behind one front door and comes out through one hub. That's the whole trick.

The front door is Caddy. One reverse proxy terminates TLS — wildcard certs issued over Cloudflare's DNS-01 challenge — and routes every request. There's nothing forwarded on my router. Instead a **Cloudflare Tunnel** dials out from the box and carries all the public traffic inward, so from the internet's point of view my home network has no open surface at all. In front of every subdomain sits one **Cloudflare Access** policy: a bypass rule for my home IPs, an email allowlist for everyone else. Private stuff stays private, and I never touch a VPN.

The hub is the portal. I got tired of remembering a dozen host:port combinations, so I pulled them all into a single dashboard (`ai.corydickes.com`) that embeds each service in an iframe and adds a control plane on top — start/stop services, drive cast devices, open VM consoles, manage guest access, run Claude Code in the browser. A wildcard catch-all means any unclaimed `*.corydickes.com` name lands on the same app and deep-links to the right tab.

The backends are deliberately boring. Most of this is plain processes under systemd's user scope, or Docker containers. No orchestrator. No Kubernetes. No cloud. Scheduling is systemd timers. State is SQLite and Postgres/pgvector. The coordination between all the AI agents editing this box is a `flock` and a git commit — not a message bus. I keep the complexity in what the services *do*, not in the platform holding them up. For a one-person system that has to age gracefully, that's the only call that makes sense.

The layers, top to bottom:

- **Edge** — Cloudflare (Tunnel + Access + DNS + wildcard certs)
- **Proxy** — Caddy, one config, every route
- **Hub** — the Spark Portal (dashboard + control APIs + Claude Code tabs)
- **Services** — ~34 systemd units, ~20 containers, ~25 timers
- **Data** — SQLite, Postgres + pgvector, a markdown note store, the QNAP NAS
- **Compute** — one box: CPU, a single GPU, and a shared unified-memory pool that every local model has to be a good citizen about

Spark Portal — one dashboard that iframes every service on my home server, and a browser terminal into Claude Code itself

The problem. My DGX Spark runs as a self-hosted home server, and over time it just kept collecting services. Open WebUI, LibreChat, SearXNG, Plex, Sonarr, Cockpit, AdGuard, VMs, cast devices, LAN gear, plus Claude Code sessions. Each one's a different port, a different URL, its own auth. Bookmarks-and-ports is a miserable way to live, and I couldn't get to any of it from my phone without punching holes in my network.

What it does. It's a single tabbed single-page app at `ai.corydickes.com` that embeds every service in an iframe and layers a control plane on top: start/stop systemd units, discover and drive Google Cast devices, poll Nest, list running VMs and open their VNC, manage Cloudflare Access guest allowlists, and — the part I'm proudest of — a full Claude Code chat tab plus a "Swarm" tab that fires N Claude workers at one coding task and shows them coordinating on a live blackboard.

How it works. The backend (`spark_dashboard.py` , ~2,900 lines) is deliberately dependency-free — Python's stdlib `ThreadingHTTPServer` , no web framework. Caddy fronts it with a Cloudflare Tunnel, so the box has zero inbound ports open; everything comes over the tunnel and is gated by one Cloudflare Access app per subdomain. A wildcard catch-all means any unmatched `*.corydickes.com` hostname (`claude.` , `swarm.` , …) serves the same portal and deep-links straight to a tab.

The clever bit is the Claude Code tab. Each chat is a detached **tmux** session (`claude-<id>`) running the real `claude` CLI on the box, and the pane's a real terminal via a per-session **ttyd** iframe. The live transcript streams over **SSE** — I poll `tmux capture-pane` on a 0.4s loop and hash the output, only pushing a frame (plain text + ANSI) when the hash changes, with a 15s keepalive comment otherwise. Because the CLIs live in tmux, sessions survive browser reloads and dashboard restarts; transcripts persist on disk, and a boot oneshot re-spawns them. `ThreadingHTTPServer` is load-bearing here so a roomful of 10-minute SSE streams doesn't block everything else.

Why it's impressive. It's a genuinely useful daily driver, not a demo — I run my whole homelab and drive Claude Code from my phone through it. And the tmux-as-durable-session-store trick is a real insight: crash-resilient, reconnectable AI chat for free, with no database and no framework.

Status. Live at ai.corydickes.com, backend :8090.

Spark Swarm — many models, one task, coordinating live instead of merging later

The problem. The obvious way to parallelize AI coding is to fan a task out to a bunch of agents, let each one go off alone, then merge. But merging N stabs at the *same* task? Most of it's throwaway, and the agents never learn a thing from each other. I wanted the opposite: several models, different tiers, all hitting one coding task *at the same time*, sorting out who does what in real time — no clobbered edits, no racing on git.

What it does. `swarm launch --task "…" --repo /path --workers sonnet,haiku` spins up N headless `claude -p` workers against one repo. Rosters mix tiers and backends: Claude aliases (opus/sonnet/haiku/fable), Grok via a local xAI shim, local Ollama, or Groq committers. Anything you list as a reviewer joins read-only — a critic that pulls the cumulative diff on a cadence and posts findings. An optional judge checks those findings against the real diff and gates approval; escalation only climbs the tier ladder when the judge says no. Operator verbs: `launch`, `status`, `board`, `logs`, `finalize`, `stop`, `cleanup`, `list`.

How it works. Every worker shares one git worktree on a throwaway `swarm/<id>` branch — nothing ever auto-merges back to the source. Coordination is an append-only blackboard, `.swarm/board.jsonl`: each post is flock'd so concurrent writes can't clobber, with typed messages (plan, claim, status, critique, done). The genuinely hard part was concurrency inside a *shared* worktree. Two fixes carry it. First, `swarm commit` is a serialized, flock'd `git add -A && commit` — raw parallel commits race on `.git/index.lock`, so that lock is basically the git version of my `coord-edit` fabric. Second, any worker that exits without posting `done` flips the run to `degraded` (you see it in `failed_workers`) so it never quietly claims `converged`. Permissions are split too: critics get a scoped read-only allowlist and can never commit; committers, in this prototype posture, run unsandboxed so they can build and test freely — the throwaway-branch worktree is the real guardrail, so the blast radius is one isolated, git-recoverable branch. Budget mode pushes the building onto off-plan lanes (Cerebras builds, Groq/Gemini review) and saves Claude for the judge and escalation.

Why it's impressive. I'll be straight: file-edit collisions are still only *softly* prevented by board claims. That's the honest Phase-1 limit. But the commit path is genuinely race-safe, and in a real run a Sonnet and a Haiku worker actually negotiated the work on the board — Sonnet withdrew a claim and took the test file — produced correct code, and passed verification. That's live multi-model coordination, not merge-and-pray.

Status. CLI `swarm`; Spark Swarm tab in the portal. Phase 1 done and tested.

Spark Diag — a nightly self-healing pass over the whole home server that fixes what it safely can and reports the rest

The problem. My Spark box runs dozens of moving parts — systemd `--user` services, system units like `cloudflared` and Plex, Docker containers, Caddy routes, a Windows VM, NAS mounts, host vitals. And stuff dies at 3 AM. A container OOMs. A CIFS mount drops. A log runs away. A cert creeps toward expiry. I didn't want to babysit any of it — and I really didn't want a static monitoring config I'd have to hand-edit every single time I stood up a new service.

What it does. Every night around 03:00, a systemd timer runs a deterministic pipeline. It discovers every target on the box, deeply diagnoses each one (liveness/resources, log/error bursts, security drift, capacity trends), then *actually remediates* the problems it's allowed to fix — restarting a dead service, remounting a dropped share, pruning dangling Docker images — and writes a narrated Markdown report plus a morning email digest. Stand up a new service and it's covered automatically. Zero code changes.

How it works. `discover.py` is the only place "what exists" gets decided: it enumerates live sources each run (`systemctl --user list-units`, watched system units, `docker ps -a`, a Caddyfile host→port parse) plus static specials — Windows VM, Plex, NAS, UniFi, host. `collect.py` gathers the raw facts, then the local `qwen3.6:35b` model triages each target in parallel into strict JSON. Cheap, and it keeps log volume off Claude. Only the flagged targets go to `claude -p` (pass 1), which emits `findings[]` and a **typed** `remediation_plan[]` drawn only from a fixed action vocabulary. And here's the key design choice: Claude never touches the system. `remediate.py` — plain deterministic Python — validates each action against an `ALLOWED_ACTIONS` whitelist, snapshots before-state, executes through a small audited handler, snapshots after, and re-verifies liveness. A restart that leaves a service worse off gets flagged loudly. A second `claude -p` pass writes the report. The kill-switch is deliberately out-of-band: if the file `REMEDIATION_DISABLED` exists, zero actions run — pure report mode — so it can't get lost in some YAML edit.

Why it's impressive. This is bounded autonomy done honestly. The LLM does what LLMs are actually good at — reasoning over messy cross-system signals and proposing fixes — but the thing that holds root-adjacent restart authority is a small, reviewable Python whitelist, not a model spitting out free-form shell. Destructive actions get escalated in the report, never executed. What you end up with is a home platform that genuinely heals itself overnight, while its blast radius stays something I can read in one file.

Status. Nightly 3 AM timer with real restart authority; report server on :8207 → portal /diag tab.

Spark Mind — a nocturnal second brain that journals my life on free-tier LLMs

The problem. I throw off a ton of exhaust — dev sessions, conversations, photos, music, messages — and none of it ever gets remembered or reflected back to me. Cloud "second brain" products cost real money per token and ship my private life off to someone else's servers. No thanks. I wanted an agent that keeps learning about me, runs on hardware I already own, and costs basically nothing to keep alive forever.

What it does. Spark Mind wakes on a timer, takes one "tick" through its senses — the box's health, what I've been working on, Spotify, conversations, photos — and asks itself whether anything genuinely new about *my life* actually happened. If so, it writes a short observation to the daily feed, and it might mint one focused curiosity to chase down later. It keeps a growing corpus of markdown notes about me, and once a day it surfaces a Facebook-style "On This Day" — memories from years past — into the feed.

How it works. It's a Python package (`spark_mind`) with a sense/cycle/tick architecture sitting on a plain markdown note store. Recall is token-overlap keyword scoring — deliberately no vector DB. The two hardest constraints were cost and memory. On cost: Claude escalation is off by default (`claude_enabled=False`) and paid Grok is hard-gated behind `allow_grok=False`, so the cascade runs free-first — Groq, then Gemini, then the local Ollama model — with a content-hashed disk cache so identical prompts never hit the GPU twice. On memory: this box has one GPU and a 121.7 GiB *unified* pool it shares with OpenClaw's pinned 35b model. Run a naive 256K context on that model and you build a ~53.6 GiB compute graph that hard-locks the whole machine. It did. Twice. So every local call is capped at 32K context, all local text generation is pinned to the *exact* model OpenClaw keeps warm — sourced straight from its keepalive payload, so the two can't drift and thrash the GPU — and the conversation drain runs sequentially. I also learned to guard the *writing*, not just the render: a hard rule in the triage prompt makes a silent tick — where only dev/ops work happened — the correct output, instead of dressing up my coding as "passive income."

Why it's impressive. It's a genuinely autonomous learning loop that's run for weeks at essentially zero marginal cost, on shared consumer hardware, without ever OOM-killing the box or leaking my data to a paid API. The engineering is in the constraints it respects, not the model it calls.

Status. Live at mind.corydickes.com and the portal /mind tab.

Spark Search — a personalization layer that quietly re-ranks my own search results

The problem. Generic search engines rank for the average person. Self-hosting SearXNG fixes the privacy side, sure — but now I'm getting the same undifferentiated results as everyone else. I wanted a search page that actually knows what I care about: the sites I live in, the projects I'm building, the conversations I've had. Without shipping any of that to a third party. And without me standing up and babysitting a whole new frontend.

What it does. For anonymous visitors it's plain, public SearXNG. For my own unlocked session it does two things. It re-orders the results by what I've actually shown interest in, and floats a "★ For you · from your history" panel of top picks. And it generates an optional AI answer, grounded in my personal data plus the live web results. Everything else — themes, autocomplete, image proxy, preferences, static assets — is the real SearXNG UI, untouched.

How it works. The gateway's a FastAPI app (`:8899`) that acts as a transparent reverse proxy. A catch-all route forwards every request to SearXNG at `127.0.0.1:8888` and hands the response straight back — carefully preserving the ~22 distinct `Set-Cookie` headers SearXNG emits on a preferences save. The one thing it rewrites is the `/search` HTML for an unlocked session: it parses the page with BeautifulSoup, reorders the `<article class="result">` nodes, and injects the panels. Any exception falls back to the untouched upstream response, so personalization can never break search.

Two signals drive it. The **re-rank** is a deterministic domain-affinity profile built from my AdGuard/DNS query logs (`dns-insights` SQLite): each registrable domain scored by distinct-days-visited times a 30-day recency decay, after I filter out the telemetry/CDN noise. Result hosts get matched against that profile — no embeddings needed at query time. The **AI answer** is real RAG. An ingest pipeline embeds my files (portal docs, this repo), LibreChat message history, and top-domain facts into pgvector (`spark-vectordb` , 768-dim `nomic-embed-text` vectors, HNSW cosine index). On `/answer` it embeds the query, pulls the six nearest personal docs, and prompts a local Ollama model with a PERSONAL CONTEXT + WEB RESULTS block. The answer card loads asynchronously, so it never blocks the results. Auth is a signed `itsdangerous` cookie issued from the Cloudflare Access identity header — one visit to `/unlock` per device.

Why it's impressive. The response-rewriting-proxy pattern is the clever bit: I inherit SearXNG's entire frontend for free, and degrade gracefully back to it on any failure. Honestly, it's a single-user tool — the re-rank is a domain heuristic, not semantic matching, and the answer runs on local hardware. But it turns passive browsing history into a search engine that's measurably mine, with nothing leaving the box.

Status. Live at search.corydickes.com; gateway :8899 fronting self-hosted SearXNG, pgvector backend.

claw (OpenClaw) — a self-hosted AI agent with a browser front door, running free cloud inference with a local GPU as its safety net

The problem. I wanted an AI agent that's always on and reachable from any browser — one that can actually run shell commands, hit the internet, and touch my home server. But I didn't want to rent a frontier API by the token, and I definitely didn't want to ship my box's contents off to somebody else's cloud. Here's the catch, though: this Spark has exactly one GPU, and a whole stack of *other* services are already fighting over it (ComfyUI, a local vision model, Spark Mind). It also sits behind Cloudflare Access, which quietly changes the auth math in ways the stock agent just doesn't expect.

What it does. claw is a self-hosted OpenClaw gateway. It runs as a systemd `--user` service on `:18789`, fronted by Caddy and a Cloudflare tunnel at claw.corydickes.com. What I get is a WebChat control UI backed by a real agent — it executes tools, it browses, and because it runs un-sandboxed on the host, it's got native full-host and full-internet access. I moved off the original NemoClaw/OpenShell sandbox for exactly that reason: its deny-by-default egress just couldn't allow-all.

How it works. Two pieces are where the actual engineering lives.

Model discipline on one GPU. Inference routes to Cerebras `gpt-oss-120b` first — free, fast, OpenAI-compatible cloud — and only falls back to a local Ollama `qwen3.6:35b` MoE when the cloud's down. So normal traffic never even touches the GPU. And the models OpenClaw is allowed to pick? That's an explicit enumerated allowlist in `openclaw.json`, not "whatever Ollama happens to have pulled" — which stops a heavy model from getting loaded by accident and OOM-ing the box. A keep-warm timer re-pins only qwen3.6:35b, and `OLLAMA_MAX_LOADED_MODELS` bounds concurrency so a local vision model can coexist and unload when it's idle. Getting here took real debugging: context-overflow crash loops, a stale per-agent `models.json` cache overriding the label, tool-calls coming out as text-JSON — all of it written up in the runbook.

Trusted-proxy auth. An OpenClaw update killed loopback auto-auth — WebSocket `token_missing`. Instead of exposing a token, I set `gateway.auth.mode: "trusted-proxy"`: Caddy injects an `X-Openclaw-User` header, and the gateway only trusts it from `trustedProxies` (`127.0.0.1`, `::1`), checked against an `allowUsers` allowlist. This is the clever bit. Cloudflare Access's home-IP *bypass* policy sends no `Cf-Access-Authenticated-User-Email` header, so I couldn't just lean on CF's identity — the Caddy-injected user closes that gap.

Why it's impressive. It's a genuinely useful agent that costs ~nothing to run, treats a single shared GPU as a first-class constraint, and gets auth right in a two-layer reverse-proxy setup where the obvious approaches silently fail. The honest "so what": this is careful plumbing, not a novel model — but the plumbing is exactly what makes a self-hosted agent survivable long-term.

Status. Live at claw.corydickes.com; gateway :18789 (Cerebras primary → local qwen fallback).

It's Giving Gone — a robot that watches half a continent of rentals and tells you when one's mispriced this weekend

The problem. Vacation rentals rot. An owner lists a cabin, the weekend gets close, it hasn't booked, and they panic-cut the price. That drop is real money — sometimes 40% — but it's invisible. It flickers onto a search page for a day and vanishes. Nobody's watching a whole region's inventory night after night, so nobody catches it. I wanted to catch it. Every time. Across every drivable getaway from a bunch of cities I care about. And I wanted the thing that caught it to be honest about *why* it thinks something's a deal, not just slap a "SALE" sticker on a normal price.

What it does. It's Giving Gone is a public site — a single-page feed of below-market rentals checking in within the next few days, a tank of gas away. The feed is dense with real product. Every deal is a card: photo, price per night, the market and dates, and a row of **deal-score explainer chips** that spell out the actual signal ("31% under its own normal rate", "dropped 18% since first seen", "22% under comparable listings"). Cards are grouped one-per-listing, so a place available for a 2-, 3-, and 4-night stay shows *once* with its best deal as the headline and the rest as a little price menu underneath — no showing you the same cabin three times. Tap a card and a modal opens with a **price-history sparkline** drawn straight from the snapshot log, min and latest points marked, so you can see the drop with your own eyes.

The filtering is deep. There's an **Area** selector (24 areas — a "local" and a "getaways" ring for each of a dozen hub cities like Charlotte, Atlanta, Syracuse, Nashville, Knoxville, Buffalo), a **Market** dropdown that narrows to that area, **Nights** (2/3/4/any), a **min-beds** stepper, a **pet-friendly** toggle, and price/score **sort**. On top of that is a whole row of **feature chips** — hot tub, pool, oceanfront, mountain view, cabin, ski, fireplace, waterfall, downtown, luxury, family, romantic — that AND together. Two of those chips are the part I'm proudest of: **Great Drives** and **Backroad & Gravel**. Cards near a legendary driving road get a badge ("🏁 near Tail of the Dragon", "🚙 near Dolly Sods"), because a chunk of the audience is enthusiasts who'll drive four hours *for the road* and need somewhere to sleep at the end of it.

That spun off its own feature: **The Drives Atlas**. A separate, curated page of every paved and gravel route I've catalogued — grouped by region, each with the town to basecamp in, a Google Maps link (with a real "turn on Avoid Highways or Google skips the whole point" warning), and a live count of how many below-market deals are near it *right now*, deep-linking back into the pre-filtered feed. It's **email-gated** — drop your address, it unlocks — which quietly makes it the lead magnet.

There's more surface than that. A live **"scanning for deals…" indicator** that lights up when a sweep is running and silently refreshes the feed when it finishes. **Email alerts** — subscribe with just an email plus your current filters, no account, and get a daily digest of the top matching drops with click-tracked Book links and one-click unsubscribe. A rotating hero hook, a tagline that rewrites itself per area so it never claims "from Charlotte" while

showing you Atlanta, "Load more" pagination, deep-linkable URL filters, and a tasteful **"tip the finder"** link that shows up *only* on Airbnb cards.

How it works. A crawler sweeps the cartesian product of `market × check-in date (today…+3) × stay length (2/3/4 nights)` and — this is the core discipline — reads prices off the **search-results grid only**, never probing listings one at a time. Search results are self-filtering: they only return available, min-nights-valid, priced cards. Each card diffs into a `price_snapshots` table keyed `(source, listing_id, checkin, nights, captured_at)`, which is what makes the whole thing a *time series* instead of a screenshot. Three signals score a deal, each with a config threshold and a composite weight: **below its own normal rate** (≥30% under a baseline — either a far-out reference sample or, cleverly, the struck-through "originally $X" Airbnb shows on a discounted card, so it fires on the very first sweep), **below comps** (≥25% under the beds-bucketed median for that market), and **price drop** (≥15% under the earliest snapshot). That last one is honest cold-start: it can't fire until there's history, and the site says so.

The scraping is the hard tier. Airbnb and Booking sit behind Akamai/PerimeterX-class walls, so `browser.py` is a stealth headless-Chromium harness driven over the DevTools protocol — real Chrome, UA auto-matched to the actual binary version (a mismatch is itself a bot tell), stealth JS that scrubs the `navigator.webdriver` tells before the site's own JS runs, consistent client hints, and a clearance-wait that distinguishes a terminal block page from an interim challenge instead of scraping the block page as if it were content. `egress.py` is a zero-config residential-proxy rotator that the browser auto-routes through the moment you drop a proxy list in place. Sources run **concurrently**, each in its own thread and DB connection, scoring **per-market as it goes** so deals surface incrementally. An adaptive scheduler thins far-out re-scans once enough history exists; a circuit breaker and time budget keep a stuck source from starving the run; the idle **local Ollama** writes a one-sentence "why it's a deal" blurb per listing (capped, cached, free); and feature tags are extracted deterministically each sweep. systemd timers sweep three times a day and mail the digest at 8am.

The hard parts. Anti-bot was the whole ballgame — and the subtle failure mode isn't a crash, it's a scraper that *silently* returns zero cards and looks like "no deals," so a lot of the engineering is about making failure loud. Concurrency over one SQLite file (WAL, busy-timeout, per-thread connections, per-instance Chrome profiles so two browsers don't collide). Deduping a listing that's live on five dates so it doesn't poison the comp median. Getting per-market incremental scoring right so the comps for a market are complete before it flushes. And keeping the whole thing *cheap* — local LLM, deterministic tagging, search-grid-only crawling.

Where it's going. The `Source` interface is built so a partner/affiliate API (Booking Demand, Expedia/Vrbo Rapid) drops in behind it — same pipeline, stabler data, and it turns Book links into commission. That's the monetization split I designed around: Booking is affiliate-able (the `/go` click-tracked redirect is already where the affiliate params will live), Airbnb isn't — hence the "tip the finder" on exactly those cards, and the email-gated Atlas building the list in the meantime.

Why it's impressive. It's a real, deployed, multi-source price-intelligence product — crawler, time-series diffing, a defensible three-signal scoring model, an anti-bot browser harness, concurrent scheduling, local-LLM

enrichment, email alerts, click funnel, and a genuinely nice SPA — and it's aimed at a niche (drive-to getaways, enthusiast roads) with an actual monetization thesis behind it. It's a startup's worth of surface area, built solo.

Status. Live at itsgivinggone.com (backend `:8214`); getaway.corydickes.com 301-redirects there.

The Distributor Desk — live wholesale cost and stock for a national IT distributor, scored for resale margin

The problem. I run gear through a reseller business. The question that actually matters every day is boring: *can I buy this part from my distributor cheaper than it sells for, after fees, with enough units in stock to fill an order?* The distributor has an API, but an API isn't an answer. Raw cost and quantity tell you nothing until you put them next to a real sell price and subtract the stuff that eats the spread. Doing that by hand, per SKU, is a non-starter.

What it does. It's a private desk that turns a major national IT-hardware distributor's live catalog into a ranked margin table. Every row is one product: our wholesale cost, total stock across warehouses, an honest resale sell-price, and the *real profit* left after resale-channel fees and shipping. Sort by best profit. Filter by category, brand, in-stock. Search by keyword, spec ("27 inch monitor," "firewall appliance"), or part number / UPC. Click a row and you get a compare modal — street prices for that exact model, a datasheet, and a blunt verdict.

How it works. The backend authenticates to the distributor's order-management API over OAuth2 client-credentials (Bearer token, tenant header), maps our tracked manufacturer part numbers to the distributor's item IDs via an item-inquiry lookup, then pulls price-and-availability in bulk (batched at the API's 50-item cap). Their `salesPrice` is our cost — that's the structural edge distribution exists to give you. A nightly crawler pages the catalog into a local SQLite store (`state/stock.db`) and stamps every material cost move into a price-history table, so a row can say "was $412, down 8% two days ago."

The clever bit is refusing to lie about the sell side. Sell-price basis is best-first: a real observed street price, else the enforced MAP floor, else the distributor's "estimated retail" — which is measured as inflated ~1.22x and, past a threshold, is thrown out entirely rather than printing a fantasy margin. Licenses and service contracts get zeroed out so an "$8k profit" renewal doesn't rank above real hardware. Demand is evidence-tiered: units-moved-in-7-days derived from stock snapshots, plus sold comps from the resale market, with unproven flips flagged as unproven. A closeout radar groups everything cheap right now by *why* — under street, under MAP, active per-unit rebate, or a recorded cost drop.

It's built multi-distributor from the start: a shared adapter contract, sibling adapters for two more national distributors staged and tested offline (waiting only on credentials), and one-row-per-product grouping so the cheapest orderable source wins. The desk itself has configurable columns — cost, retail, MAP, real profit, margin %, floor, rebate, stock spread, cost trend, shipping, weight, warranty, TAA, origin — all sortable.

Why it's impressive. It's a real procurement tool, not a demo. It talks to a live enterprise distributor API, normalizes it, and layers in the judgment a buyer actually applies — and the judgment is conservative on purpose. Most "arbitrage" tools overstate profit because they trust a made-up MSRP. This one would rather say "I don't know the sell price" than flatter you. That honesty is the whole value.

Status. Private, Cloudflare Access-gated; backend on `:8213`. In active use against the distributor's live catalog API.

Self-Maintaining Storefronts — a catalog that re-derives itself every night, and won't claim anything it can't prove

The problem. A mid-size dealer's website is usually a museum of decisions somebody made once. Ninety thousand products, hand-keyed, on a stack older than most of the staff. Prices drift from what the distributor actually charges. Items sit "in stock" for a year after the line went end-of-life. Nobody is lying — it's that keeping ninety thousand rows honest by hand is not a job a person can do, so it silently stops being done, and the site quietly becomes fiction.

What it does. It takes a snapshot of the catalog once, and after that it rebuilds itself. Every night it asks the distributors what those parts cost and how many are on the shelf, compares them, sources from whichever can actually ship, and prices the result — never below a manufacturer's minimum advertised price. Then it writes down what changed since yesterday: prices that moved, items back in stock, products that no longer exist. Nobody hand-keys a price, ever again.

How it works. Three packages that talk only through two SQLite files: a one-time harvest, a nightly pipeline, and the storefront. The pipeline is the interesting one — it quotes every credentialed distributor for every part, keeps every quote rather than just the winner, and applies pricing rules in a fixed order so the same inputs always produce the same page.

Two things I'd defend in an interview. The ordering path is built end to end — draft, approve, submit — and is **structurally unable to place an order**: the credentials load from a directory that is empty, and five separate conditions must all be true to transmit. You can demo the refusal on screen. And there's a claims linter that fails the build if the site asserts anything the data doesn't support. If the pipeline can't establish stock, the page can't say "in stock." Not "shouldn't" — can't.

The second one proved it was a product. The first build was one dealer in one industry. The second was a different industry entirely, on a different website platform, and the architecture held: one adapter file per platform, one row per dealer. That one started as an audit — what the site *claims* against what it can actually establish — and the audit is now the opening move, because it's a better sales conversation than a demo. You don't argue that the rebuild is worth it. You show them the gap and let them decide.

Why it's impressive. Not the scraping, which is ordinary, and not the storefront, which is a website. It's that the catalog cannot drift from reality, because reality is the only place it comes from. Every number on the page has a provenance, and anything without one doesn't render. That's a different class of artifact than a site somebody maintains carefully — carefully is a promise, and this doesn't need one.

Honest caveat. It re-derives what the distributors know. If they carry bad data, the site carries it faithfully and confidently, which is worse than carrying it uncertainly. The changelog is the mitigation — a wrong number that moves is visible, and I'd rather see a diff than trust a feed.

Status. The customer-agnostic core is extracted and standalone: stdlib only, no third-party dependencies, 129 tests. Dependency-free on purpose — this gets handed to a customer's own environment eventually, and "it needs a package manager" is a conversation I'd rather not have.

Movement II · Professional Engineering

The software I've built for a managed service provider — production tools that run a real business, used daily by a real team. Client, employer, and vendor names are redacted; the scale and the outcomes are intact.

Multi-Tenant Firewall Monitoring Platform — real-time security posture across 100+ firewalls, without drowning anyone in alerts

The problem. We managed firewalls for dozens of client organizations, and the vendor's own cloud console was built for someone who owns one network. Per-device views. No cross-tenant rollup. Nothing that answered "which of my clients is on fire right now?" So techs checked boxes one at a time, or didn't check at all, and we found out about problems when a client called. It was a mess.

What it does. It polls every firewall we manage on a continuous cycle and pulls posture into one place: interface and WAN health, config drift, blocked-traffic patterns, bandwidth, licensing, and rule changes. It ingests syslog, scores traffic for possible data exfiltration, snapshots each device's config and diffs it against the last known-good, opens tickets automatically when a WAN circuit drops, and serves all of it through a tabbed web dashboard with per-client views.

How it works. Python, about 33,000 lines across 91 modules. A poll loop fans out across tenants with a worker per client and a cooldown map, so a flapping device can't spam anyone. Syslog lands in sharded SQLite with write-ahead logging, and bandwidth gets pre-rolled into summary tables, because querying raw syslog to render a dashboard tab is a great way to build something nobody wants to load. The UI is a `dashboard/` package where each tab is its own module emitting its own markup, CSS and JS — that's what keeps a big dashboard from collapsing into one unmaintainable template.

Two choices I'm actually proud of. First: the firewall API allows one admin session at a time, so naive polling kicked human engineers out of the console mid-change. A monitoring tool that fights your own team is worse than no monitoring tool. I built three-layer gating that detects an active operator and backs off. Second: alert suppression. Early on it alerted on every rule change, every cycle, forever. I added sticky per-finding state plus a minimum re-alert floor, so you hear about something once when it's new and not again until it means something.

Why it's impressive. Honestly, the polling wasn't the hard part. It was everything around it — vendor APIs that return HTTP 200 with the real failure buried in the response body, high-availability pairs that misreport which member is active, and the fact that alert fatigue silently destroys a monitoring system's value. Getting the noise floor right took more iterations than the features did. It's also genuinely multi-tenant, which means every query, view and alert has to be correctly scoped or you leak one client's data into another's dashboard.

Status. In production. Runs as a managed service, polls continuously, and the dashboard is what people actually open in the morning. Still the system I extend most often.

Fleet Bootstrap — rebuild an entire server fleet from one executable and a passphrase

The problem. Everything we'd built ran on a single server. Services, scheduled jobs, a public tunnel, a dozen automations, all held together with secrets and symlinks and service accounts accumulated over a year. If that box died, the honest recovery estimate was "several days and a lot of guessing." Nobody had written it down. Here's the thing about that kind of risk: it's invisible right up until it isn't.

What it does. Takes a bare Windows Server with nothing on it and reinstalls the entire fleet — every application, service, scheduled task, symlink, service account, secret, and the tunnel configuration — back to working state. Run it, walk away, come back in 15 to 30 minutes to a functioning box.

How it works. Idempotent numbered steps: prerequisites, service accounts, drive layout, clone applications, build virtual environments, symlinks, secrets, service registration, scheduled tasks, tunnel. A `bootstrap.ps1` orchestrator runs them in order, and any step runs standalone. If it dies halfway, re-run it. Nothing breaks on a second pass.

The state it restores isn't hand-maintained, which matters more than it sounds. `manifests/` holds captured snapshots of the real fleet — services, symlink targets, accounts — and a scheduled sync job refreshes them from the live box. The harness can't drift away from reality the way written documentation always does.

Secrets are the interesting bit. They live in the repo encrypted with git-crypt, so the recovery artifact is self-contained. That creates a bootstrap paradox: you need the decryption key to recover, and the key was on the box that just died. I solved it by compiling a launcher, `recover.exe`, with the GPG private key embedded at build time via `build.ps1`. Recovery becomes one file plus one passphrase — no separate key file to keep in sync, no chicken-and-egg. The trade-off is real and I documented it rather than hiding it: that executable is now sensitive, so it lives in access-controlled storage, and if it ever leaks the play is rotate the key and rebuild before anyone can brute-force the passphrase.

There's also `validate.ps1`, a read-only health check, because a DR harness nobody tests is a DR harness that doesn't work.

Why it's impressive. Most people write a recovery runbook. Runbooks rot. This one executes, it's been validated, and it self-updates from live state. The single-file model is the part I'd defend in an interview — a deliberate security trade-off with the reasoning and the mitigation written down, not a shortcut taken quietly.

I'll be straight about the limits too. It restores infrastructure, not data; application state comes from backups. Credentials bound to a specific machine identity can't transfer and have to be re-issued after recovery. Knowing what your recovery plan doesn't cover matters as much as knowing what it does.

Status. Built, committed, validated and documented. The recovery artifact is stored outside the fleet it recovers, which is the only place it's any use.

Ops Digest Pipeline — a dozen vendor APIs, one place to look, and nothing shouting twice

The problem. An MSP runs on other people's platforms. Endpoint detection, backup, SaaS backup, network monitoring, wireless, MFA, cloud identity, email security — each with its own portal, its own idea of "urgent," and its own daily email. Nobody logs into twelve consoles. So real problems sat unseen in dashboards while inboxes filled with noise nobody read.

What it does. Each platform gets an integration that pulls its state on a schedule, works out what actually changed or genuinely needs a human, and emits a digest: failed backups, offline devices, coverage gaps, expiring credentials, risky configuration. Findings flow into one internal web hub where someone acknowledges them or turns them into a ticket with a click.

How it works. Hub and spoke. Each spoke is a self-contained scheduled job — mostly PowerShell, some Python — that authenticates to its vendor, normalizes results, and publishes a baseline of findings. The hub is a Flask app with a registry mapping each source to its label and state directory, so adding an integration is a config entry and a job, not a refactor.

The piece that makes the whole thing work is suppression. The hub exposes a suppressed-findings API and every spoke calls it before sending. Acknowledge something once and it stops appearing in tomorrow's digest, across every source, without anyone editing a script. Before that, each digest re-reported the same known-and-accepted issues on every run, which is precisely how people learn to ignore automated email.

The unglamorous work was authentication. Twelve platforms, twelve auth models — OAuth client credentials, basic auth, partner-scoped tokens delegating into customer tenants, keys with source-IP restrictions. Several return HTTP 200 with the real failure in the body. Several rate-limit hard enough that a fast loop looks like an outage. So every integration is paced, retried, and scope-verified by confirming the data actually came back for the tenant I asked for. Silent fallback to the wrong tenant is the failure mode you never catch by checking status codes, and it's the one that quietly puts one client's data in another's report.

Why it's impressive. It isn't one clever thing, it's twelve unglamorous things that stayed running. Each integration is small. The value is that they're consistent — same scheduling, same credential handling, same suppression contract, same delivery path. That consistency is what let the collection grow to a dozen without becoming a dozen snowflakes each with its own failure mode.

Look, the honest caveat is that a pipeline like this is only as good as its signal-to-noise ratio, and tuning that is permanent work, not a launch task. I've re-tuned thresholds repeatedly and I expect to keep doing it.

Status. Running on schedule in production, feeding a hub the team uses daily. New integrations get added on the established pattern.

Fleet Control-Enforcement Audit — the security feature that was installed everywhere and working nowhere

The problem. We maintained a shared blocklist of malicious domains and addresses that every managed firewall pulled automatically. Add a bad domain centrally, every client is protected. That was the design, anyway. Then a domain I'd added didn't block, at a site where the feed was definitely configured. That's the kind of thing you can't leave alone.

What it does. Audits every firewall in the fleet and answers one question honestly: if I add an entry to the shared blocklist, will this device actually deny the traffic? Not "is the feed configured" — whether the control is wired end to end.

How it works. The block only functions if three things line up: the feed materializes into an address object, that object is nested inside a wrapper group, and an enabled deny rule references that group. Break any link and the feed keeps downloading happily forever while enforcing nothing. So the audit walks all three across every device through the vendor's management API, resolving group membership transitively rather than trusting names.

Here's the part worth telling. My first run said 71 of 106 firewalls were broken. That number was wrong, and I'm glad someone pushed back instead of letting me act on it. I'd only queried IPv4 objects. At most sites the wrapper group and its rule are IPv6 objects even though the members are IPv4 — an artifact of how the control was originally deployed. My check false-negatived every correctly configured site.

The corrected audit checks both address families. The real answer was 16 genuine gaps, not 71. I later found two more traps the same way: sites name the feed object differently, so any name-based check false-negatives them, and group members nest under a different key than the obvious one, which makes a properly populated group look empty. Every one of those bugs failed in the same direction — reporting a working control as broken.

Why it's impressive. The engineering here is straightforward. The judgment is the point. A security audit that cries wolf is worse than no audit, because the remediation is going and changing enabled deny rules on production firewalls at client sites. Acting on a false positive means causing an outage to fix a problem that was never there. I now treat "this control isn't working" as a claim that needs a second, structurally different confirmation before anyone touches a device, and I'd rather report an honest UNKNOWN than a confident wrong answer.

The other lesson is the one I'd lead with: an unenforced control is more dangerous than a missing one, because everybody believes they're covered.

Status. The audit tooling is read-only and re-runnable, the genuine gaps were identified and largely remediated, and the methodology traps are written up so the next person doesn't rediscover them the hard way.

Time Entry Review — catching the billing mistakes before the invoice goes out

The problem. Engineers log time all day, fast, between other work. Some entries land blank. Some go against the wrong client. Some are three typo-riddled words that a customer will eventually read on an invoice and question. Nobody was reviewing them, because reviewing hundreds of entries by hand is nobody's idea of a job. So mistakes went out the door, and the ones that came back came back as billing disputes.

What it does. Scans time entries from our PSA and flags the ones a human should look at: empty or near-empty narratives, spelling and grammar problems, entries booked against a company that doesn't match the work described, and durations that don't look right. Then it presents them in a review UI where you fix an entry inline or accept a suggested correction with one click.

How it works. Python and Flask, around 5,000 lines. A scanner pulls entries over the PSA's REST API. A classifier in `classify.py` scores each one across the failure categories. Typo detection was the fiddly part — off-the-shelf spellcheck drowns in technical vocabulary, hostnames and product names, so `mine_typos.py` mines the corpus itself to learn which oddities are normal here and which are genuine errors. Findings surface in a review UI backed by a small state store, and corrections write back through the API.

One design decision I'd defend. Every flagged row gets a manual edit box, not just an "apply suggested fix" button. I originally shipped one-click-apply only, for the categories where the fix looked safe. That was wrong. The suggested correction is sometimes incomplete and sometimes just off, and the reviewer needs to be able to write the right thing rather than accept the machine's guess. Automation you can only accept or ignore quietly trains people to accept.

Why it's impressive. It's a small app pointed at a real money problem. Billing errors cost twice — once in the write-off, and again in the trust you spend explaining them. It's also the kind of automation people actually adopted, because it puts a human at the decision point instead of silently rewriting their work.

The honest caveat: it's a detector, not an oracle. It surfaces candidates and a person decides. That's deliberate, but it does mean the value depends on someone running the review.

Status. Deployed as an internal web app with scheduled scans, used as a pre-invoice check.

Effort Attribution Analysis — the number was off by an order of magnitude, and finding out why was the whole job

The problem. Leadership needed to know how much engineering time one platform was really consuming across the client base — the kind of number that decides whether you keep paying for something. Easy question. The obvious query produced an answer that was wrong by roughly ten times, and it looked entirely reasonable.

What it does. Attributes labor hours to a topic across the full ticket history — tens of thousands of tickets, about 15,600 of which carried real logged labor — and reports a defensible range instead of one confident number.

How it works. The naive approach is: find tickets mentioning the thing, total their hours. That returned 13,610 hours. The defensible figure was 1,444. Two traps caused the gap, and neither is visible unless you go looking.

First, engineers log entire days against a handful of annual catch-all tickets. One alone carried 2,554 hours across 568 entries. A single passing mention in one entry drags the whole ticket in. The fix was to stop attributing per ticket and start attributing per time entry: split each narrative into work items, compute what share of the entry actually concerns the topic, count the whole entry when the topic dominates it, and pro-rate when it doesn't — floored at the minimum billing increment.

Second, substring matching on short acronyms. A plain "contains" filter for a three-letter product name matched hostnames that merely contained those letters, dragging in dozens of unrelated monitoring tickets. The fix is word-boundary matching, plus actually reading the resulting title list before trusting the population.

A couple of other things would have quietly skewed it. Project-based labor carries a task reference and no ticket reference, so joining only on tickets drops real hours into a null bucket. And the fair denominator for "share of our support load" is tickets that consumed labor, not all tickets — most are automated alerts no engineer ever touched.

Why it's impressive. Anyone can run the query. The skill was distrusting a plausible answer, finding the two structural reasons it was inflated, and reporting a low/central/high range with the rule that produced each figure, so the person making the decision could see the uncertainty instead of inheriting false precision.

Honestly, the most valuable output wasn't the number. It was writing down the traps, because those same two will corrupt every future "how much time did we spend on X" question anyone asks of this dataset.

Status. Delivered with documented methodology. The attribution approach is reusable for any topic against the same data.

Contract & Pricing Generator — a full sales document set in minutes instead of an afternoon

The problem. Every new client agreement meant someone opening last month's contract, saving-as, and hand-editing names, dates, terms and per-seat pricing into a Word document. Then rebuilding the pricing spreadsheet to match. Then filing both somewhere the rest of the company could find them. It took most of an afternoon, and copy-paste editing means stale values survive in documents nobody re-reads. Wrong pricing in a signed contract is an expensive typo.

What it does. You fill in a web form — client details, term, service tiers, seat counts, rates. It produces the finished agreement as a Word document and a matching pricing workbook, consistent with each other by construction, and files both into the company document library automatically.

How it works. A small Flask app built from a few focused modules. `pricing.py` holds the rate logic and is the single source of truth for every number that appears in either artifact — that's the whole trick. `docxgen.py` renders the agreement and `xlsxgen.py` builds the workbook, and both read from the same computed pricing object, so the contract and the spreadsheet cannot disagree. `graphfiling.py` handles the authenticated upload into the document library, so filing isn't a manual step someone forgets on a Friday.

Two things I learned the hard way. Generated spreadsheets need computed values in total cells, not formula strings — the library that writes the file doesn't evaluate formulas, so totals render blank until someone opens the sheet and triggers a recalculation. To whoever receives it, that just looks broken. And there's a `test_generation.py`, because document generators fail in the ugliest possible way: they produce a file, it opens fine, and one field is quietly wrong.

Why it's impressive. It isn't architecturally exciting and I won't pretend otherwise. What makes it good is that it eliminated an error class rather than just saving time. Deriving both documents from one pricing computation means the failure mode where the contract says one number and the spreadsheet says another is gone. Not less likely. Gone.

It also landed inside the workflow people already had. Output goes where the documents already live, in the formats the business already uses. Nobody had to adopt a new system, which is most of the reason anyone used it.

Status. Running as an internal web tool. Generates and files complete document sets on demand.

Shared Engineering Knowledge Base — turning tribal knowledge into something version-controlled

The problem. A small team accumulates an enormous amount of undocumented knowledge. Which API lies about success. Which client's setup is deliberately weird. Which "problem" is actually working as designed. It lived in people's heads and in chat history. New engineers rediscovered the same traps, and so did experienced ones six months later at 11pm.

What it does. A version-controlled knowledge base covering platform and API access, infrastructure conventions, per-project state and gotchas, client-specific quirks, and cross-platform identity mappings — roughly 7,500 lines across topic-scoped documents, plus the tooling that keeps it honest.

How it works. Plain Markdown in git, one document per domain, loaded on demand rather than all at once. The tooling around it is what makes it stick. A commit helper records each edit attributed to whoever made it, so every change is logged and every prior version is recoverable. A scheduled job auto-commits anything left uncommitted and mirrors history to storage the team can't overwrite, so a change can't be lost to someone's mistake. And a suggestion script gives people a path for "I think this is wrong, but I'm not confident enough to rewrite shared truth" — the suggestion queues for review instead of being dropped or, worse, applied on a guess.

There's a credential helper in there too, abstracting over per-engineer secret stores with a shared machine-scope fallback. Scripts stopped hardcoding one person's profile path, which is what previously made half our tooling run correctly for exactly one person.

The rule I care most about is verify before you edit. It's shared truth for everyone, so confirming a claim live beats pasting a guess. I've corrected entries that turned out to be wrong, including my own.

Why it's impressive. It's the least glamorous thing here and probably the highest leverage. Documentation projects usually fail because writing is a favor you do the future at your own expense. This one survived because the tooling made contributing cheap, attribution automatic and mistakes recoverable — and because it's structured for lookup at the moment of need, rather than onboarding-day reading nobody opens twice.

Honest caveat: a knowledge base is only true on the day it's written. Entries go stale and I've been burned trusting one that had drifted. The mitigation there is culture, not tooling — verify live before acting on anything load-bearing.

Status. Actively maintained and used daily, with full change history, a review workflow and automated off-box backup in place.

Movement III · The Foundation

Before the tools, the trade. Four years of hands-on operations and consulting across dozens of client networks — around 7,700 logged hours of edge, servers, security, migrations, and onboardings. This is the delivery work the software grew out of, and the reason it solves the right problems.

Network Edge Program — four years of keeping dozens of client networks connected and defended

The problem. Every client has an edge, and the edge is where the business actually stops working. Firewalls reach end of support. Circuits fail at the worst possible time. A growing office outruns the hardware it was sized for five years ago. Across a client base this adds up to a permanent rolling program, not a project — and if you treat it as a series of unrelated emergencies, you spend your life doing emergency work.

What I did. I ran edge infrastructure across the client base: firewall replacements and upgrades, high-availability pair builds, new internet circuit turn-ups and failover configuration, site-to-site connectivity between offices, remote access, and wireless. Roughly 970 hours across 827 logged efforts — the single largest theme in four years of my work.

The approach. The pattern that made this sustainable was treating replacements as scheduled lifecycle rather than reactive break-fix. Track support expiry across every device, size the replacement against actual throughput rather than the last model, and stage the config offline so the cutover window is short and reversible.

The genuinely hard parts were never the configuration. They were the cutovers. Replacing firewalls across multiple sites for one client meant sequencing so that no two sites were down at once and the site-to-site tunnels still came up when the peers were mismatched mid-migration. High-availability pairs are worse than they look — a badly built HA pair fails over into an outage instead of preventing one, so I made verified failover testing part of the build rather than something we'd "check later." Circuit turn-ups meant coordinating a carrier who doesn't work your hours against a business that can't be down during theirs, which is mostly a scheduling and communication problem wearing a technical costume.

Why it's impressive. Honestly? Because most of it was invisible. A firewall migration that goes well produces no story — the office comes in Monday and nothing is different. Four years of that, across dozens of organizations, weekend cutovers included, with the failures rare enough to count. That's not a clever architecture, it's operational discipline, and it's the thing clients actually pay for.

It also gave me something I couldn't have got from reading: a real feel for how network gear lies to you. Devices that report healthy while dropping traffic, HA members that disagree about which one is active, vendor management consoles that show stale state. That instinct is what later made the monitoring platform worth building.

The record. Ran as a standing program across the client base. The build and cutover standards I settled on are still how new sites get done.

Security Incident Response — the calls that start with "something's wrong" and end with a written timeline

The problem. Small and mid-sized organizations get hit, and they don't have an incident response team. They have their MSP. So when ransomware fires on a file share at 9pm, or an endpoint agent flags a threat nobody recognizes, or a domain controller starts doing something it has no business doing, the phone rings and you're it. There's no playbook handed to you and no second shift.

What I did. I led incident response engagements across the client base: ransomware detections, endpoint threat containment, a domain controller credential-theft indicator worked under a formal statement of work, and remediation programs following third-party penetration tests. Around 105 hours of explicitly incident-tagged labor, though the named engagements each ran for weeks once you count the follow-through.

The approach. The order matters more than the tooling. Contain first — isolate the affected endpoints before you start being curious, because the investigation is worthless if the thing is still spreading while you do it. Then establish scope: what else did that account touch, what else did that host talk to, and how far back does this actually go. Only then remediate, and only then write it up.

The domain controller case is the one I'd talk about. An alert fired on abnormal access to the directory database — the file that holds every credential hash in the domain. That's either a catastrophe or a backup job with bad manners, and those two look identical in a single alert. The work was proving which, by reconstructing what process touched it, under what account, on what schedule, and whether that pattern had a legitimate explanation. Getting that wrong in either direction is expensive: cry wolf and you burn a client's trust and a weekend, miss it and they lose the domain.

The other recurring lesson was that the loudest indicator usually isn't the important one. Endpoint tools flag plenty of things. The signal is in what's *around* the detection — the logon that preceded it, the account that shouldn't have been used, the host that has no reason to be talking to that one.

Why it's impressive. Look, I'm not going to claim to be a dedicated IR specialist. What I can claim is that I've done this under real pressure, with a business waiting, without a team behind me, and produced a defensible written account each time — timeline, scope, what we did, what we couldn't determine. That last part matters. The honest "here's what we cannot prove from the evidence available" is the hardest paragraph to write and the one that earns trust.

The record. A recurring responsibility across four years, with several formal engagements delivered end to end and documented.

Datacenter & Server Lifecycle — replacing the machines a business runs on, without stopping the business

The problem. Servers age out. Storage fills. A drive goes into predictive failure on a Friday. Underneath every client's day-to-day is physical and virtual infrastructure that somebody has to plan, replace, migrate and occasionally rescue — and unlike a workstation, you can't just swap it and apologize.

What I did. Server and storage lifecycle across the client base: replacement server builds and migrations, SQL Server and remote desktop host deployments, network-attached storage to SAN data migrations, hardware failure recovery, Active Directory domain controller work, patch infrastructure, power and UPS, and an in-place Linux distribution upgrade on a production application server. Around 515 hours across 374 efforts.

The approach. The template for a server replacement is dull on purpose: build alongside, migrate data in stages with verification at each one, cut over in a defined window, and keep the old system intact and startable until the new one has proven itself through a full business cycle including month-end. That last bit is the part people skip. Something always only breaks at month-end.

Two jobs stand out. A storage migration moving a client off a NAS onto SAN storage — the interesting problem there isn't the copy, it's permissions and open files. Data that arrives with mangled ACLs is technically migrated and practically useless, and anything with a lock on it silently doesn't come across unless you're checking. The other was upgrading a production Linux server across major distribution versions, in place, for an application whose vendor support matrix and the OS's supported upgrade path didn't quite agree. That one needed a tested rollback more than it needed a clever plan.

Hardware recovery came up more than I'd have liked. Drives in predictive failure, a NAS that lost its array, a client environment that had to be brought back into production after a serious failure. Those teach you fast that the backup you never tested is a rumor.

Why it's impressive. This is the work where mistakes are least recoverable. You can roll back a config change. You can't un-migrate data you corrupted or un-lose an array you rebuilt wrong. Doing it repeatedly across a client base without a data-loss incident isn't glamorous, but it's the credential that matters for anyone touching production infrastructure.

It's also where I learned to distrust green checkmarks. A backup job reporting success and a backup you can actually restore from are different claims, and only one of them is worth anything at 2am.

The record. Delivered continuously across four years. The staged-migration-with-verification approach is still my default for anything touching production data.

Client Onboarding & Provider Transitions — inheriting environments nobody documented

The problem. When a client moves to you from another provider, you inherit an environment with no documentation, no credentials, no asset list, and often an outgoing party with zero incentive to help. Sometimes there's real friction about handing over data at all. Meanwhile the client expects support to work on day one, because from their side they just changed vendors.

What I did. I ran onboardings and provider transitions: discovery of what actually exists, credential and data recovery from outgoing providers, bringing environments up to our monitoring, backup and endpoint protection standards, and building the documentation that should have existed already. Roughly 198 hours across 148 efforts, including a data preservation engagement that ran the better part of a year.

The approach. Discovery first, and assume the asset list you were handed is wrong. Scan the network, reconcile against what the client believes they have, and expect a gap — there's always a server nobody mentioned. Then get control: administrative access, DNS and domain registrar control, backup and licensing ownership. Control of DNS and the domain registrar is the one people underestimate, and it's the one that hurts most when a transition sours.

Then standardization: monitoring agent, backup coverage, endpoint protection, patch policy, documentation. The point isn't the tools, it's that every environment converges on the same shape so the team can support any client without learning a snowflake.

The data preservation engagement was the hard one — extracting and preserving a client's data from a departing provider's systems over an extended period, where the constraint was cooperation and access rather than technology. Most of that work was patience, sequencing, and writing down exactly what had been recovered and verified at each step, because "did we get everything?" is a question you need an evidenced answer to, not an opinion.

Why it's impressive. Transitions are where an MSP relationship is won or lost, and they're the least controlled work there is. You're operating in an environment you didn't build, with information you can't fully trust, against a deadline set by someone else's contract end date.

Honest caveat: no onboarding I've done was clean. There's always a discovery gap, and something surfaces in month two. What I got good at was surfacing those early and saying so, rather than finding out in front of the client.

The record. Delivered across multiple client transitions. Discovery, control and standardization sequence is a repeatable playbook.

Monitoring & Tooling Standardization — replacing "we'll notice eventually" with something that tells you

The problem. Monitoring coverage across a client base tends to grow by accident. Something gets deployed at one site, someone else's preferred tool at another, and a legacy platform nobody wants to touch sits in the middle. The result looks like coverage on paper and isn't. Nobody can answer "are we actually watching this client?" without going and looking.

What I did. I led rollouts and consolidation of the platforms the team relies on: replacing an aging network monitoring system with RMM-based monitoring, deploying a network discovery and monitoring platform across client sites, rolling out DNS-layer filtering, standing up automated network device configuration backup, and auditing endpoint protection deployment for coverage gaps. Around 550 hours across 496 efforts.

The approach. Every one of these rollouts followed the same arc, and the arc is mostly not technical. Establish what coverage exists today — honestly, which usually means the answer is worse than assumed. Pick the standard. Deploy in waves rather than all at once, because the first wave always teaches you something that changes the runbook. Then, and this is the step people skip, go back and verify deployment actually landed on every target rather than trusting the console's install count.

The endpoint protection audit is the clearest example. The management console reports what's checked in. It says nothing about the machines that should have an agent and don't. Reconciling the deployment list against the actual asset inventory is where the real gaps live, and it found them.

Replacing the legacy network monitor was the one with the most organizational friction. The old system worked, sort of, and people knew it. Migration meant rebuilding checks, mapping alert thresholds so the new platform didn't either flood or go quiet, and running both in parallel long enough to trust the replacement. Turning the old one off too early would have meant a blind spot nobody announced.

Why it's impressive. Here's the thing about monitoring rollouts: everyone counts them done at "deployed." Deployed is the halfway point. Done is when the alerts are tuned enough that people believe them, and coverage is verified against reality instead of the vendor's dashboard. The gap between those two states is where most monitoring programs quietly die.

This is also the work that directly seeded everything in the professional-engineering section. Doing coverage reconciliation by hand, repeatedly, is precisely what convinced me it should be automated.

The record. Platforms deployed, standardized and verified across the client base. Coverage-verification-against-inventory is now how I judge whether any rollout is finished.

Cloud and Identity Migration — moving workloads and access into Microsoft's stack, carefully

The problem. Clients kept arriving at the same crossroads: aging on-premises servers, a mail platform overdue for migration, remote work that the existing access model was never designed for. The cloud answer is obvious in a slide deck and considerably less obvious when it's a real business with a line-of-business application, a directory built up over a decade, and no appetite for downtime.

What I did. Azure server deployments and cloud backup design, mail migrations into Microsoft 365 and the network and firewall changes those require, identity and conditional access configuration with the documentation to go with it, and integration work joining our documentation platform to delegated cloud administration. Around 456 hours across 450 efforts.

The approach. For workload moves the decision I pushed hardest on was what *not* to move. Lifting a server into cloud hosting because it's aging often just relocates the problem and adds a monthly bill. The useful version of the conversation is which workloads genuinely benefit — the ones that need availability the client's server room can't provide, or that are already talking mostly to cloud services — and which should stay put or be replaced with a SaaS equivalent.

Mail migrations were the recurring set piece, and the technical work is less than half of it. Cutover timing, mobile device reconfiguration, and the fact that mail flow changes ripple into the firewall and the security stack — inbound routing, allowed senders, whatever's scanning the mail — mean the coordination is the job.

Identity is where I got most careful. Conditional access is a genuinely powerful control and an extremely effective way to lock an organization out of itself. A policy that looks correct in the portal will happily strand the administrator who wrote it. So: exclusions for break-glass access first, report-only mode before enforcement, and written documentation of what each policy does and why, because the person debugging a blocked sign-in in a year won't be the person who built it. I later hit the flip side of this professionally — a client's device-compliance policy blocking legitimate service-provider automation — which taught me the same lesson from the other direction.

Why it's impressive. The restraint, more than the builds. Saying "this workload shouldn't move" to a client who's been told everything should go to the cloud is a harder conversation than doing the migration, and it's usually the right one.

The record. Delivered across multiple client environments. Report-only-then-enforce and documented break-glass access are non-negotiable defaults for me on any identity change.

How It's Built — Working With AI, Not Just Using It

Everything in this document was built by one person. That sentence is only true because of *how* I work — and honestly, this is the chapter I actually want you to remember.

Most people use AI like a smarter autocomplete. One prompt, one answer, one file at a time. I use it as **labor I direct** — often several agents at once, running against real repositories, coordinating with each other, checking each other's work — and I've had to build real infrastructure to make that safe. The tools are out there; anyone can grab them. Running them continuously, as the normal way you ship? That's still rare. That's the part worth writing down.

Parallel by default. When a task splits into independent pieces, I don't grind through them in order — I hand one to a subagent, spin up the next, and integrate what comes back. This document was built exactly that way: a swarm of agents each read one project's source and drafted its write-up while I wrote the connective tissue. Minutes, not an afternoon. The discipline that makes it work is simple — split by *file* or subsystem, so two agents never fight over the same lines.

Coordination is a lock and a commit, not a framework. Multiple Claude sessions edit this box at the same time, so a naive "last write wins" would quietly destroy work. My answer is deliberately low-tech: `coord-edit`, a little wrapper that grabs an exclusive `flock` on the repo, atomically swaps the file, and commits the change tagged with whichever session made it. Collisions show up as git conflicts instead of just vanishing. It's a dozen lines of shell doing the job a lot of people reach for a database to do.

Swarms take it further. For one genuinely hard task I'll launch a *swarm* — several models of different tiers hitting the same problem at once, in a shared throwaway git branch, negotiating who does what on an append-only blackboard, with commits serialized through that same `flock`-and-git idea. Cheaper models build; a stronger one judges; escalation only kicks in when the judge rejects the work. It's the `coord-edit` philosophy, scaled up from *me and my sessions* to *a team of models*.

AI that runs the box, not just writes it. Some of these systems *are* the AI, working on its own. Spark Diag diagnoses and repairs the whole machine every night — with the twist that the model only ever *proposes*; a small, reviewable Python whitelist is the only thing with restart authority. Spark Mind learns about my life on a nocturnal timer. The pattern I keep coming back to is **bounded autonomy**: let the model reason freely, but put a deterministic guardrail between its judgment and anything you can't undo.

Sustainable by construction. One box, one GPU, a shared memory pool, and a flat refusal to burn money. Local models do the cheap high-volume grunt work; free-tier cloud APIs (Cerebras, Groq, Gemini, Grok) get routed to by cost and capability; the strongest paid models are saved for the high-value, interactive moments. Several of these things run indefinitely at essentially zero marginal cost — and that's a design achievement, not an accident.

That's the method. The projects are just what it produces.

Appendix — The Complete Index

The professional work lives in Movements II and III above. This appendix is the home platform's long tail — the services, products, and supporting repositories beyond the flagships. Each line is its own git repository.

AI systems & agents

- **Spark Mind** — nocturnal learning agent that journals my life on free-tier LLMs *(flagship)*
- **Spark Swarm** — multi-model coordinated coding swarm *(flagship)*
- **Spark Diag** — nightly self-healing diagnostic + guardrailed remediation *(flagship)*
- **claw / OpenClaw** — self-hosted browser-based tool-using agent *(flagship)*
- **Spark Diary** — read-only timeline of how the box evolves; git-log + topology snapshots, qwen draft → Claude-verified chapter summaries (:8211)
- **spark-bounty-stack** — scheduled recon/scan/triage over authorized public bug-bounty programs; hybrid local-qwen → Claude triage
- **spark-internal-scan-stack** — design sketch: internal authorized-network vuln scanner (inverted allow-only scope)
- **certforge-stack** — automated IT-cert study-product factory (generate → adversarially verify → package)

Web products & personalization

- **Spark Portal** — the hub that iframes and controls everything *(flagship)*
- **Spark Search** — personalized re-ranking layer over self-hosted SearXNG *(flagship)*
- **It's Giving Gone** — last-minute rental-deal finder, itsgivinggone.com *(flagship)*
- **The Distributor Desk** — private, Access-gated procurement/pricing desk over a national IT-hardware distributor's catalog *(flagship)*
- **mcwatch** — price/stock watcher against defended retail targets; the shared pipeline behind It's Giving Gone and the Distributor Desk
- **dns-insights** — DNS analytics dashboard: collector + alerts + LLM-narrated daily digest + blocklist recommender (:8200)
- **mcclaskey-beef** — DTC beef storefront pitch prototype over a lot-level traceability model + claims linter

Media stack

- **sonarr-stack** — TV PVR (:8989)
- **threadfin-stack** — IPTV → HDHomeRun bridge (:34400) + nightly playlist refresh
- **searxng-stack** — self-hosted metasearch engine (fronted by Spark Search)
- *(also running: Plex, LibreChat, Open WebUI, Music Assistant, n8n, Portainer, neko)*

Infrastructure & plumbing

- **spark-home-lib** — shared building blocks (ACL, workspace+trash, SSE poller, integrations) used across projects
- **adguardhome-stack** — local DNS server (:53) + UI (:3000), AdGuard-DNS-cloud DoT upstream
- `.config/caddy` — the one Caddyfile that routes every subdomain (Tunnel + wildcard certs)
- `.config/systemd/user` — all `--user` service + timer units
- **rar2fs-setup** — FUSE mount to stream RARs off the QNAP NAS without extracting
- **coord-edit** — the `flock` + atomic-rename + git-commit wrapper that lets many AI sessions edit this box safely

Cross-cutting capabilities (not a repo — a posture)

- **Free-tier model routing** — Cerebras / Groq / Gemini / Grok + local Ollama, chosen by cost and capability, paid Claude reserved for high-value interactive work
- **Cross-machine bridges** — a Windows VM and a work box reachable and driveable over SSH + tmux
- **Zero open inbound ports** — everything public arrives over a Cloudflare Tunnel, gated by Cloudflare Access