Adventures in Ham Radio AI

Real-World Artificial Intelligence for the Amateur Radio Operator

Jonathan W. Pearce, WB2MNF

Adventures in Ham Radio AI

Real-World Artificial Intelligence for the Amateur Radio Operator

All projects described in this book are real. Every example reflects actual work done by the author. The book itself was drafted by Claude AI under the author's direction and editing; the Preface was written by the author.

Gloucester County Amateur Radio Club (W2MMD)

Version 6.0 · Published September 7, 2026

This book is revised as new projects are completed. The version number and date above identify this release; check the club website for the current edition.

Preface

This is the only part of this book that I personally wrote in entirety. Everything else was written by Claude with extensive prompting from me. I wrote it because I believe that many hams don't use AI programs because they don't know of enough reasons to do so. They don't have examples of what it can do, they've heard the stories about hallucinations and don't want to get wrong answers – or they simply don't know where to start.

Having worked with AI programs since mid-2025 I'm in the opposite corner – I use them for everything. If I listen to an interesting podcast I'll dump it into Claude and ask for a summary. If I get a medical test I'll paste the test results into Grok or Claude and ask what they mean and how they compare to previous tests. Before going on vacation I'll have Claude check out the dive operators and spots and give me a plan for the week. If I want to learn about something (EME operation, how WSJT works, how to decode the ISS DATV signals) I'll ask Claude to construct a summary or an entire curriculum to teach me. For some users it's not knowing where to start – for me it's not knowing how to stop.

Believing that the best way to introduce AI to the ham community is to give examples, I've tried to fill this book with as many diverse uses as would be helpful to readers. Seeing many different examples may trigger some opportunities for readers to try Grok or Claude or ChatGPT on some ham activity. Once you've gotten the hang of it you'll find other uses in your life. Your personal productivity will increase significantly – you can find stuff out instantly, start working on tasks where you don't have sufficient background letting the AI teach you or handle the unknowns, and sometimes remove boring tasks that you never wanted to do anyway. I suddenly discovered a whole world of projects that I wanted to do but couldn’t dream of in the past either because I lacked the knowledge or the time needed to complete them that were now entirely possible, and I’m finding new ones every day.

Having Claude write the book - essentially about himself – also meant that he could use information his own examples. You’ll see that in the detail – he actually went back into the projects that he’s describing for the details about what he and I each did. I didn’t have to remember anything other than the project itself – I simply told him to write about developing the Space Camp documentation or decoding the ISS DATV signals and he went back into his memory, dug up the details and wrote it all himself. (Yes, I know I’ve crossed that anthropomorphizing-AI line, but it seems right…)

Feel free to skip around as you read – there are many sections that may not be applicable to some readers so find something interesting and dig into it. Then go try it yourself.

73 de Jon WB2MNF

Contents

Introduction

Why This Book Exists

The biggest obstacle to using AI is not cost, complexity, or technical skill. It is a lack of imagination. Most people try AI once or twice, get a decent answer to a simple question, think "that's neat," and stop. They never discover that the same tool that answered their question could also write a program for them, teach them a subject they've never studied, design a system they couldn't build alone, or operate their equipment while they sleep. They stop not because AI failed them but because they never thought to ask for more.

I wrote this book to fix that. Every chapter is a real example of something AI did for me or for the GCARC Skunkworks group—not a hypothetical demonstration but an actual project with actual results. Some are simple: drafting a newsletter article, reviewing a friend's car repair bill, editing an email. Some are substantial: building a web application I couldn't have coded myself, configuring servers over SSH without touching a keyboard, processing a raw radio signal into watchable video. The goal is not to teach you how to use any particular AI product. The goal is to show you enough different applications that your own imagination starts filling in the gaps with projects that matter to you.

What I've Learned About Using AI

The skills to use AI well develop with use. There is no manual. But I can offer two pieces of advice that would have saved me weeks if someone had told me at the start.

First: talk to it as if you were talking to a person. Not a computer, not a search engine—a person. Forget that it's AI. If you were sitting next to a knowledgeable colleague and needed help, you wouldn't type three keywords into their forehead. You'd say "I'm trying to match a 50-ohm feedline to an end-fed dipole—what do I need to build?" AI responds to the same thing: context, specificity, and clear intent. The more naturally you talk to it, the better your results will be. Keep talking to it as you work, the way you'd keep a running conversation going with an assistant sitting next to you. Tell it what you're seeing, what you're confused about, what changed since the last time you asked.

Second: if you don't know how to start, ask the AI how to start. This sounds circular but it is the single most useful prompt pattern I've found. I watched a STEM student freeze completely when I put a small Estes rocket on the table in front of him and asked him how high it would fly. He had no idea where to begin. I told him to type into Claude: "Jon put a rocket in front of me and asked me how high it would fly. How do I figure that out?" Claude immediately walked him through the steps—identify the motor, look up its total impulse, estimate the rocket's mass, apply the equations. Once he knew how to start, he could do the rest. The barrier was not knowledge. It was imagination.

That is the barrier this book is designed to break.

The examples in this book are drawn from ham radio because that's the audience. But AI does not care about domains. I've used it to produce detailed analyses of healthcare test results, helping me understand what my numbers meant and what questions to ask my doctor. I used it to review a friend's proposed service bill from a car dealer—Claude identified nearly a thousand dollars in unnecessary upsells, including a GDI fuel system service being charged for an engine that isn't direct-injected. I've used it for tax planning, investment analysis, and curriculum design. The ham examples are here because they're the most relevant to this audience, but every pattern in this book transfers to anything else you need done.

One use deserves a special mention because it changes what's possible for anyone willing to sit down and ask questions: AI as a personal tutor. It will teach you any subject, at whatever level and pace you set, without ever getting impatient—and when you're done it will turn the conversation into a presentation, a study guide, or a set of test questions. I lean on this constantly — the two chapters of Part 3, Learning With AI, are devoted to exactly how it works.

How This Book Was Made

Claude wrote almost this entire document. The preface is mine. Everything else—every chapter, every appendix, every revision—was drafted by Claude and edited by me. The book you are reading is itself an example of the process it describes.

The development spanned many working sessions over weeks. Chapter drafts went through two to five revision cycles each as I corrected facts, adjusted tone, added material from voice notes, and pushed back on sections that were too wordy or missed the point. The total time Claude spent generating and revising this content is measured in hours; the total time I spent reviewing and directing it is measured in days. That ratio—AI generating at speed, the human editing with judgment—is the ratio that every project in this book operates on.

How to Use This Book

You do not need to read this book cover to cover. Skim the chapter titles, find the areas that are relevant to your interests, and dig into those. If you're curious about writing and document production, start with Part 4A. If you want to automate things with code, jump to Part 4C. If you want to understand what an AI agent is and whether you need one, read Chapter 3 and then decide.

If you have never used a Raspberry Pi and the idea of typing commands into a Linux terminal is intimidating, read Appendix A on the Raspberry Pi and then Chapter 4 on Claude Code. The combination may change your mind. Code can do almost anything on a Pi—install software, configure services, debug errors, write and deploy programs—and it does all of it over SSH without you typing the commands yourself. For someone who has been interested in Pi-based ham radio

projects but put off by the Linux learning curve, Code opens doors that were previously closed. You describe what you want. Code does the rest.

If you are already comfortable with Pis and programming, the project chapters will give you ideas for things you haven't tried yet.

Whatever your starting point, the underlying message is the same: AI is not a search engine, it is not a toy, and it is not going away. It is a tool that rewards curiosity, and the best way to learn what it can do is to ask it.

Part 1 · Foundations

Chapter 1 — What Is AI and Why Should a Ham Care?

A year ago, I couldn't have written this book. Not because I didn't know enough about AI — I didn't, but that's not the reason. The reason is that the things described in this book — having an AI configure servers, write programs, translate codeplugs, debug rotator controllers, process satellite signals, and generate an entire technical website — were not things I believed a ham radio operator could do with a conversation. I thought AI was a clever search engine. I was wrong by a large margin.

I started with Grok, the AI built into X (formerly Twitter), and used it for months. Grok was capable and I used it for dozens of applications: interpreting medical test results, designing a healthy breakfast smoothie, drafting documents, answering questions. I used it to build the GCARC event registration system, a substantial web application that tracks attendance at club functions — 862 lines of HTML and JavaScript, with Firebase authentication and a Firestore database backend — all vibe-coded by describing what I wanted without understanding the underlying languages. Grok also taught me how to set up the Firebase database and how to use its management tools. With a SuperGrok subscription, I never ran out of tokens. Grok was a genuinely useful tool.

But Grok had a ceiling, and I hit it in two places. First, the Technician license practice test system. I needed the AI to ask multiple-choice questions and, when the candidate got one wrong, to explain why the answer was wrong without revealing the correct answer. Grok couldn't do it. Every time the candidate chose the wrong answer, Grok would say "No, A is wrong. The correct answer is C" — which is exactly what a test system must not do. No amount of reprompting fixed it. Second, when I tried to use Grok as the language model behind OpenClaw, the AI agent platform, the hallucinations were so severe that I nearly gave up on OpenClaw entirely. The agents would fabricate data, invent commands that didn't exist, and confidently report results from tasks they hadn't actually performed.

I switched to Claude, and the problems disappeared. Claude handled the license test logic on the first try. When I pointed OpenClaw at Claude's API instead of Grok's, the hallucinations vanished and the agents started performing reliably. From there, the scope of what I asked Claude to do expanded rapidly: writing articles, designing infrastructure, configuring servers, generating presentations, processing signals, teaching me subjects I didn't know. Within a few months, almost everything new on the GCARC Skunkworks website was AI-generated, and I had two AI agents running autonomously on my home network.

I should be clear: the fact that this book uses Claude throughout does not mean other AI systems aren't capable. There are many AI labs producing excellent products, and what works best for you may depend on your specific needs and preferences. But all of the examples in this book are from Claude, so I can vouch

for its capabilities and not for others. If you're using a different system and getting good results, keep using it.

Knowledge is Irrelevant

Here is the single most important thing I want to say in this book, and I'm saying it in the first chapter so it frames everything that follows: from now on, knowledge of how to do things on a computer – and in many other areas - is largely irrelevant. The AI knows how to do it. It can write the program, configure the server, generate the document, process the data, design the system. What matters now is not knowing how to do the work but seeing that the work can be done and asking AI to do it.

This is not an exaggeration. Marc Andreessen, the venture capitalist who co-founded Netscape and runs Andreessen Horowitz, recently described a phenomenon he calls "AI vampires" — programmers who are so productive with AI coding tools that they stop sleeping. They stay up all night, running multiple AI agents simultaneously, building programs they've thought about for years but never had time to write. Non-programmers at his own firm who had never written a line of code are now, in his words, "ripping out software like crazy" using AI agents.

That's Silicon Valley. In ham radio, the version is quieter but no less real. I'm not a software developer. I'm a ham radio operator who wanted to automate some things at the clubhouse. Claude Code wrote the programs. I'm not a web designer. Claude wrote the Skunkworks website content. I'm not a curriculum developer. Claude helped design a STEM satellite communications curriculum, reviewed it for cross-session consistency, and produced student workbooks, instructor guides, and slides.

The skill that matters now is not programming, not web development, not system administration. It's the ability to see what AI can do for you and make it work. That's why I wrote this book: to present as many real examples as possible, drawn from actual ham radio projects, to stimulate your imagination about what AI can do for your station, your club, and your operating.

What AI Actually Is

When most people hear "artificial intelligence," they picture robots or sentient computers. The AI that's relevant to this book is nothing like that. It's a large language model — an LLM — which is a program that has been trained on an enormous body of text and learned to produce useful output from natural language input. In practical terms, it's a program you talk to. You type a question or request in plain English, and it responds in plain English. If you ask it to write Python code, it writes Python code. If you ask it to explain how FT8 packs a message into a fifteen-second transmission, it explains it. If you ask it to design a lightning protection plan for an amateur radio station, it designs a lightning protection plan.

Tokens: Why You Should Care

A language model doesn't process words. It processes tokens, which are fragments of text — sometimes a complete word, sometimes a syllable, sometimes a single character. The word "transceiver" is two tokens. The callsign "WB2MNF" is three or four tokens depending on the model. As a rough guide, one thousand tokens is about seven hundred fifty words, and a page of text is about four hundred tokens.

You should understand tokens for three reasons, all of which affect your daily use of AI.

Tokens measure AI thinking. Everything the model reads and everything it writes consumes tokens. A long, detailed prompt with an uploaded document uses more tokens than a short question. A long response costs more tokens than a short one. When you give Claude a twenty-page PDF and ask it to analyze the content, that's roughly eight thousand tokens of input before it even starts thinking about its answer.

Tokens are the unit of subscription limits. Most AI services have usage caps tied to tokens, and this affects ordinary chat use, not just agents. With my Claude Pro subscription at $20 per month, I regularly hit my token limit during a morning of heavy use and was told to wait several hours for it to refresh. The frustration of being mid-project and locked out drove me to upgrade to a Claude Max subscription, which has a higher ceiling but still runs out during sustained heavy sessions. By contrast, my SuperGrok subscription never hit a limit. This is a real consideration when choosing a platform.

Tokens are the unit of API billing. If you run an AI agent that makes API calls, you pay for every token it consumes — which makes the choice of model a cost decision, not just a quality one. That principle is easy to state and surprisingly easy to ignore; I learned it the expensive way with one of my own agents, a story I tell in full in Chapter 3. The short version of the lesson: use the cheapest model that produces acceptable output for each task.

The Context Window

The context window is the total number of tokens the model can see during a single conversation. It includes everything: system prompts, conversation history, uploaded documents, and the model's own responses. Claude can handle roughly two hundred thousand tokens — equivalent to a four-hundred-page book. This is usually more than enough for any single conversation, but if you're working on a complex project over many exchanges, the context will eventually fill and the model will lose track of earlier details. Starting a new conversation with a summary of where you left off is the standard workaround.

System Prompts

A system prompt is a set of instructions given to the model before the conversation starts. It's like a job description: it tells the model who it is, what it's supposed to do, what constraints to follow, and what tone to use. The user never sees the system prompt, but it shapes every response.

System prompts are central to several projects in this book. The FCC license training system uses one that tells the model to act as a patient tutor and never reveal correct answers. The STEM student deployment uses one that restricts the model to STEM topics and logs bypass attempts. My production agent's propagation report uses one that specifies the style and data sources for band conditions reporting. Writing a good system prompt is a skill covered in Chapter 2.

Cloud Models and Local Models

There is no single AI. There are many models, built by different companies, with different capabilities and different ways to access them. They fall into two broad groups.

Cloud models run on the provider's servers. You interact with them through a web interface or an API, and the computation happens remotely. The major cloud models include Claude (Anthropic), GPT (OpenAI), Grok (xAI), and Gemini (Google). Each has different strengths, and the landscape changes rapidly. Claude is what this book uses, but the concepts transfer to any platform.

Local models run on your own hardware. I bought my Mac Mini to run them. You download the model files, install a serving program like Ollama, and run inference on your own machine. The computation never leaves your network, there are no per-token charges, and the model is available even without internet. The tradeoff is capability: a model small enough to run on affordable hardware is less capable than a large cloud model. Local models mostly become relevant when you start running AI agents and want to control cost, so I'll save the detail — which hardware, which models, which tasks — for Chapter 3, where the agents that use them are described.

What AI Is Good At

AI is genuinely useful for a wide range of tasks that matter to ham radio operators. It can write and edit text — articles, reports, presentations, newsletters. It can generate code — Python programs, JavaScript functions, configuration files, shell scripts. It can teach and explain any technical subject at any level with infinite patience. It can transform structured data — codeplugs, CSVs, configuration files. It can administer systems — install software, configure services, debug errors, all over SSH. And it can plan and analyze — drafting multi-document plans, reviewing work for consistency, identifying gaps, and serving as a devil's advocate.

Every one of these capabilities is demonstrated with a real project later in this book.

What AI Is Bad At

Knowing what AI can't do reliably is equally important.

Guaranteed factual accuracy — AI models occasionally produce confident, plausible, and completely wrong statements. This is called hallucination, and checking for it is not optional. If Claude tells you a specific frequency is allocated to a specific service, or a component has a specific rating, verify it against a primary source. The model is not lying; it's producing the most probable response, and sometimes the most probable response is wrong. This was the problem with Grok in OpenClaw — the hallucinations were frequent enough to make the agent unreliable.

Real-time data — a language model's knowledge has a cutoff date. It doesn't know today's propagation conditions or current TLE data unless it has access to a search tool.

Precise measurements — AI cannot read your SWR meter or detect a signal on your SDR. It can analyze data you've already captured, but it cannot interact with physical instruments.

Visual layout — Claude can write the logic behind a Node-RED flow but cannot produce a visually coherent graphical layout. Any task where spatial arrangement matters more than logic is a weakness.

Four Ways to Interact with AI

This book covers four distinct ways that ham radio operators interact with AI. Each has different strengths and use cases, and it's worth fixing the differences in your mind now because the rest of the book uses these terms constantly.

Chat

The simplest mode. You open claude.ai in a browser, type a question or request, and get a response. Chat is the right tool for learning, writing, brainstorming, planning, and reviewing. The WSJT seminar preparation, the CrossTalk articles, the EmComm plans, and the license training system were all developed through chat. Chat is also where you start when you don't know how to start: "I want to do X — how should I ask you to help me with that?"

Claude Code

Claude Code runs in your computer and has access to your local file system and, with SSH keys configured, other machines on your network. It doesn't tell you what commands to run — it runs them itself, reads the output, and iterates until the task is complete. Code is the right tool for programming, system administration, and debugging, and Chapter 4 is devoted to it.

Claude Cowork

Cowork is a desktop application built for multi-step knowledge work — tasks that involve several files, several sources, and a sequence of operations. Where chat is a conversation and Code operates on your machines through a terminal, Cowork works directly with documents: reading them, editing them, checking its own output, and correcting itself without being prompted at each step. The monthly meeting deck automation in Chapter 8 is the clearest example. Cowork is newer than the other three modes — some of the projects in this book were built through chat simply because Cowork didn't exist yet, and would be done in Cowork today.

Agents

An agent is an AI instance that runs persistently on your hardware, executing tasks on a schedule without you being in the conversation. One of my agents generates propagation reports every three hours and publishes them to the club website autonomously. Agents use platforms like OpenClaw and Hermes, covered in Chapter 3.

What It Costs

AI is not free, but the cost structure is more flexible than most people assume.

Claude Pro subscription: $20/month. This gives you access to claude.ai chat and Claude Code. For many users this is sufficient, but be aware that heavy use will hit the token limit, at which point you're told to wait several hours for it to refresh. If you're the kind of person who spends a full morning working with AI — and you will be, once you see what it can do — you'll hit that wall regularly.

Claude Max subscription: higher ceiling. I was forced by my own impatience to upgrade to Max after repeatedly hitting the Pro limit mid-project. Max has a substantially higher token allowance but can still run out during a sustained heavy session.

Claude API: pay per token. If you run agents that make API calls, you pay for the tokens consumed. This is where costs can sneak up on you, as Chapter 3 describes.

Local models: free at inference time. Once you've bought the hardware, running local models through Ollama costs nothing per query. For routine tasks that don't need cloud-model capability, this is the economical path — again, more in Chapter 3.

What Comes Next

This chapter gave you the conceptual foundation. You know what an LLM is, why tokens matter for usage and billing, the difference between cloud and local models, what AI does well and where it fails, and how chat, Code, and agents differ. None of this requires you to be a computer scientist. It requires you to be a ham radio operator who's willing to type a question and see what comes back.

Chapter 2 teaches you how to ask good questions — the prompting skills that determine whether you get a generic answer or a genuinely useful one. From there the book moves through the tools (agents and Claude Code) and then into the projects, each one drawn from real work at the GCARC Skunkworks lab. If you want the hardware background that many of those projects assume, Appendix A covers the Raspberry Pi.

Everything in this book actually happened. Nothing is hypothetical. And it all started with a question typed into a text box.

Chapter 2 — Ask AI Anything: Prompting for Real Results

I used to think AI was a fancy search engine. You asked it a question, it gave you an answer, and you decided if the answer was good enough. That mental model sells it short by an enormous margin. What I've learned over the past year of using Claude daily — for ham radio projects, club administration, infrastructure work, and teaching middle school STEM — is that AI is less like a search engine and more like a very patient, very knowledgeable collaborator who never gets tired of your questions.

This chapter is about how to think about AI before you type your first prompt. The mindset matters more than the technique.

Your First Instinct Is Wrong

Most people approach AI the way they approach Google: form a reasonably specific query, submit it, evaluate the result. That works, but it's leaving most of the value on the table.

The better approach is to start with what you're trying to accomplish, even if you don't know how to accomplish it. Don't research first and then ask. Ask first. Use AI to orient yourself before you decide what to do.

Here's a concrete example. A club member wants to build an EFHW matching transformer. The Google approach: spend 45 minutes reading about toroids, figure out the terminology, then maybe ask a more specific question. The AI approach is to simply describe the goal: "I want to match a 50 ohm feedline to an end-fed half-wave antenna. How do I build a device to do that? Give me exact specifications."

That's it. No prior research required. Claude will ask clarifying questions if it needs them, state its assumptions clearly, and give you a complete buildable answer: core type, turns count, wire gauge, test procedure. If you happen to already have an FT-240-43 core and #14 AWG wire, adding that detail will tighten the answer further. But you don't need it to start.

The rule: describe your goal. Let AI figure out the method.

AI as Tutor

The most powerful thing I've done with AI isn't building tools — it's learning. It is an infinitely patient tutor that will teach you any subject at whatever depth you ask for, and never makes you feel foolish for asking the same thing five different ways until it clicks.

Better still, the deliverable comes out of the same session: when you're done learning you can ask it to turn the conversation into a presentation, a study guide, or a set of test questions. I lean on this so heavily that the two chapters of Part 3 — learning the WSJT digital modes well enough to teach them in Chapter 5, and

working out a satellite's footprint geometry in Chapter 6 — are given over entirely to it. The habit to build now, while we're on the subject of prompting, is simple: when you want to understand something, open a conversation and keep asking until it clicks.

Don't Limit the Ask

The other instinct that works against you is scoping down your request to something that feels "reasonable." I caught myself doing this constantly in the early months. I'd think "I can't ask it to do that" and then ask for something smaller. Sometimes I'd think that I needed to Google something before giving AI a command. Stop doing that.

Ask for the whole thing. If you want to translate a DMR codeplug from one radio to another, say so. If you want to automate the IC-9700 to switch VFOs on command, say so. If you want to build an entire license training system, say so. The ceiling is higher than you think, and the only way to find it is to bump into it.

And if you're not sure how to frame the ask — if the problem feels too big or too vague to put into words — there's a prompt for that too: "I want to do X. How should I prompt you to help me with that?" Ask AI how to ask AI. It works.

The Revision Loop

One more mindset shift: stop trying to write the perfect prompt on the first try. Each conversation turn is cheap. Perfectionism up front is a trap.

A recent example: the CrossTalk article announcing the EFHW Tech Saturday went through three revision cycles in about four minutes. The first draft was good but used the word "matchbox" (too informal), referenced the Skunkworks group by name (premature), and buried the take-home theme. Three turns of "change this, drop that, tighten this section" and it was done.

The instinct is to get it right immediately. The better approach is to get something on the page and iterate. You'll get there faster.

What This Looks Like in Practice

Across the past year, I've used AI to learn WSJT digital modes well enough to teach them; to design and write the content for the WSPR Network automated propagation reports; to build a Technician license training system for middle school students; to translate DMR codeplugs between radio families; to draft EmComm capability plans, meeting decks, and club governance documents; to write Python programs for radio control, satellite tracking, and data processing; and to configure Linux servers, set up services, and debug system errors — without touching a keyboard.

None of these started with research. All of them started with a question.

Part 2 · The Tools

Chapter 3 — AI Agents: OpenClaw, Hermes, and Persistent Automation

Chapter 1 introduced three ways to work with AI: chat, Claude Code, and agents. Chat and Code both require you to be present — you start the conversation, guide it, and end it. An agent is different. An agent is an AI that runs persistently on your own hardware, executing tasks on a schedule or in response to triggers, without you being in the conversation. It works while you sleep, while you're at your day job, and during Field Day when you're too busy running stations to check a dashboard.

I run two openclaw agents at home. Their names are Bob and Arnold, and they are deliberately opposite to each other — different hardware, different models, different jobs, and different priorities. Between them they illustrate almost everything you need to know about running AI agents, so this chapter introduces them once and refers back to them everywhere else in the book.

What an Agent Needs

Every agent rests on three things: a machine to run the agent software, a language model for it to think with, and a network connection between them. The agent software is lightweight — essentially a scheduler and a message router — so it runs comfortably on a cheap computer. The language model is where the real work happens, and it can run locally on your own hardware or in the cloud behind an API. Choosing which is the central decision in running an agent, and it's where Bob and Arnold part ways.

Agent Platforms: OpenClaw and Hermes

An agent needs a platform — software that handles the scheduling, the connection to a language model, the system prompt, and the management of tasks. The two platforms I've worked with are OpenClaw and Hermes.

OpenClaw is the one I use for both Bob and Arnold. It's a third-party AI agent product — an Ollama partner — that runs on a Raspberry Pi or any Linux machine. You configure it through a single JSON file, openclaw.json, which specifies which language model to use, where to find it on the network, what system prompt to give the agent, and what tasks to run on what schedule. OpenClaw can point at a local model running under Ollama or at a cloud API like Claude's. That flexibility is its great strength — and, as the token-cost story later in this chapter shows, its most expensive trap.

Hermes is another agent platform with a similar concept but a different architecture. I have less direct experience with it, so I won't pretend otherwise — the important point is that the agent idea isn't tied to one product. The

fundamental shape is the same across all of them: a persistent process with a system prompt, a model connection, and a task schedule.

Bob: The Personal Platform

Bob is mine, for me. He runs on a Raspberry Pi 5 and his job is the personal, low-stakes work where it's perfectly fine if an answer is occasionally wrong — summarizing things for me, drafting throwaway text, running experiments. Bob exists so I can find out what purely local, free models can actually do.

That's the defining rule for Bob: he uses only local models, never a paid cloud API. Cost is paramount; accuracy is negotiable. His current model is qwen3.6:35b-a3b-nvfp4, served from my own hardware on the LAN (see the model server below). Because nothing Bob does is mission-critical, he's the perfect test bench — I try a new model, a new system prompt, or a new scheduled task on Bob first, watch how a free local model handles it, and only promote it to production if it earns the trip. If Bob gets something wrong, I learn something and nothing breaks.

Figure 1 - Bob

Arnold: The Production Workhorse

Arnold is the opposite of Bob in every way that matters. Arnold is production. He posts content to the Skunkworks website and answers WSPR Network queries — work that real people read and rely on. For Arnold, failure is not acceptable. A

garbled propagation report or a hallucinated answer on the club site is worse than no report at all.

So Arnold's rule is the inverse of Bob's: accuracy is paramount, and cost is a secondary concern. Arnold runs on Claude — specifically Claude Haiku and Claude Sonnet, chosen per task. Routine, well-defined jobs run on Haiku, which is fast and inexpensive; the work that demands careful reasoning or interpretation runs on Sonnet. He does not use free local models, because the point of Arnold is to be right.

Arnold currently lives on a Dell R530 rack server, but that's a historical accident I'm undoing. The R530 has 256 gigabytes of RAM, which sounds ideal for running models locally — but it has no GPU (Graphical Processing Unit, the device that does AI reasoning), and that turns out to be the thing that matters (see the hardware discussion below). Since Arnold uses Claude's API rather than a local model, the big server is pure overkill, so Arnold is migrating onto a Raspberry Pi 5. The agent software is featherweight; only the model needs power, and Arnold's model lives in the cloud.

The Model Server and the Hardware That Matters

Bob's local models have to run somewhere, and that somewhere is a Mac Mini M4 Pro. It runs Ollama bound to the network so its models are available to any agent on the LAN. With 48 GB of unified memory it can host sizeable models — currently qwen3.6:35b-a3b-nvfp4, the model Bob uses.

Why a Mac Mini rather than that big Dell server? Because language-model inference is not primarily a RAM problem — it's a GPU problem. Modern LLMs are built to run on graphics processors, which do the massive parallel matrix math of inference far faster than a CPU. The R530 has plenty of RAM and no GPU, so running a model on it is painfully slow: minutes per response where the Mac Mini answers in seconds.

This is worth understanding because it tells you what hardware to buy. If you already have a gaming PC with a high-end graphics card — an NVIDIA RTX 3080, 3090, or 4090 with 12 to 24 GB of VRAM — you already own excellent hardware for local models. The same GPU that renders games at high frame rates is exactly what accelerates inference. Gamers have a real advantage over typical hams here. If you don't have a GPU, Apple Silicon Macs (M1 through M4) are the exception that makes a Mac Mini such a good model server: their unified memory architecture lets the CPU and GPU share one pool of RAM, so they run sizeable models at conversational speed without a separate graphics card. That's why I bought the M4 Pro for exactly this purpose.

The Token Cost Lesson

This is the most expensive mistake I made with AI, and it's the practical payoff of the token discussion in Chapter 1. When I first set Arnold up, I pointed every task at Claude Sonnet — scheduled reports, routine status posts, short summaries, log parsing, everything. Each call cost a few cents. Arnold ran six scheduled jobs a day plus triggered tasks, and the per-call cost was invisible. The cumulative cost was not: over a few weeks it added up to enough money to buy a basic HF transceiver. I didn't notice for too long because the outputs looked great and no single charge was alarming.

What finally caught it was, ironically, Claude itself. I asked it to read Arnold's OpenClaw logs, total the tokens by job type, and produce a cost breakdown. The report was clear and damning: routine jobs that didn't need Sonnet's reasoning were burning Sonnet-priced tokens simply because I'd never told Arnold to do anything else. The fix matched the philosophy that already separates Bob and Arnold: route each task to the cheapest model that still meets its accuracy bar, and also eliminating unnecessary jobs that were fun but not needed and that burned a lot of tokens. Arnold's routine jobs moved to Haiku — much cheaper, still Claude-accurate — while the analytical work that genuinely needs it stayed on Sonnet. Work where being wrong is acceptable doesn't belong on Arnold at all; that's Bob's department, running a free local model. The cost dropped immediately.

The lesson has three parts. Know which model each task uses and why. Use the cheapest model that produces acceptable output for that task. And monitor your usage — if you can't see how many tokens your agent is consuming and on what, have Claude build you a dashboard that shows it.

What Arnold Actually Does

Arnold's flagship job is the WSPR Network propagation report. Every three hours — 6 AM, 9 AM, noon, 3 PM, 6 PM, and 9 PM Eastern — he pulls data from two sources: the NOAA Space Weather Prediction Center (solar flux, K-index, A-index, geomagnetic conditions) and WSPRnet spot data from wspr.live (actual spots received worldwide in the preceding window). He feeds both to the model with a system prompt that tells it to behave as a propagation observatory — reporting what the data shows, not forecasting what might happen — and publishes the plain-English band-conditions summary to the club site at wspr.wb2mnfai.org via the WordPress REST API. He also drives the AI interpretation layer on the WSPR participant dashboard and several scheduled content updates on the Skunkworks website. A human could do this in thirty minutes, six times a day, every day, with no weekends off. Arnold does each cycle in about ninety seconds and never complains.

Getting Data Out of an Agent

An agent runs in the background with no screen, so how do you see what it's doing? The simplest answer I've found: have the Pi run a lightweight web server. Claude wrote a Python Flask dashboard for Bob that serves over the home LAN, showing current status, a token-usage thermometer, a 24-hour graph of calls by job type, and a log of recent tasks. I open a browser to Bob's address and see everything. For finished work — reports, summaries — an agent can publish directly to a website via the WordPress REST API (Arnold's method) or to Google Docs, or just write files you can reach over the network. And if you don't know how to set any of that up, ask Claude; if you give Claude Code SSH access to the Pi, it will install and configure it for you, as the next chapter describes.

When You Need an Agent — and When You Don't

Most AI tasks don't need an agent. An agent earns its keep only when all three of these are true: the task recurs on a schedule, it can be fully specified in advance with no human judgment needed mid-task, and the output has to be produced whether or not you're available. Arnold's propagation report meets all three. If instead you need a back-and-forth conversation, that's chat. If you need to build or configure something, that's Claude Code. Most hams should start with chat, move to Code when they want to build, and only reach for an agent when they catch themselves doing the same scheduled AI task over and over and wishing it would just happen on its own. Setting one up — the machine, the model, the openclaw.json — is walked through in Chapter 13.

Chapter 4 — Claude Code as Operator

There's a version of using AI where you ask it for instructions and then go do the work yourself. That's useful. There's a better version where the AI does the work directly. This chapter is about that second version — Claude Code as an operator that reaches into your machines and does the job — and it sits here, after the agents chapter, because some of the most striking examples involve Code operating the very agents you just met.

What Claude Code Is

Claude Code is a different tool from the claude.ai web interface. It runs in your computer, and it has access to your local machine: it can read and write files and run commands. With SSH keys configured, it can also reach other machines on your network. SSH — Secure Shell — is the standard way to log in to a remote computer and run commands on it over the network; it's how you talk to a Raspberry Pi sitting headless on a shelf, and it's covered in more detail in Appendix A. The important point here is that once Claude Code can SSH into a machine, it can operate that machine as fully as you could sitting at its keyboard. Claude Code at my house is installed on Bob’s Raspberry Pi and can operate on any Pi on my local network or the clubhouse network.

The Old Way: Paste and Pray

Before I understood this, every AI-assisted setup job followed the same tedious loop. Claude tells me what to do, I type it, something fails, I copy the error message, I paste it back to Claude, Claude suggests a fix, I type that, something else fails. Repeat. It works — I got things running this way — but it's slow, and every transcription is a chance to introduce a new typo and a new error. The human is reduced to a slow, error-prone relay between a chat window and a terminal.

The Breakthrough: Code Operates the Machine

The shift came when I set up the Mac Mini as a model server and needed to reconfigure both of my agents — Bob and Arnold, described in Chapter 3 — to point at it. Instead of relaying commands by hand, I gave Claude Code SSH access to both machines, told it what I wanted, and got out of the way.

Code SSH'd into the first agent and read its existing openclaw.json configuration. It identified the model-provider settings and updated them to point at the Mac Mini's Ollama endpoint. It hit a dependency error on the Node.js version, read it, diagnosed it, and resolved it without asking me. It edited the systemd service file so the right environment variables were set at startup, restarted the service, ran a test prompt, and confirmed the model was reachable. Then it repeated the whole process on the second machine, adapting for its different software version. I did

not type a single command or paste a single error message. The job took less time than the first failed attempt had taken the old way.

That's the difference between an advisor and an operator. Code didn't tell me how to fix the agents; it fixed them.

The Same Pattern, Everywhere

Once you've seen Code operate one machine, you start handing it everything, and most of the projects later in this book are examples of it. Code wrote the CI-V program that turns on the IC-9700's second VFO and deployed it to a spare Pi (Chapter 14). It built the entire WSPR beacon SD-card system on a test Pi and automated the card-burning (Chapter 12). It compared a working and a broken Node-RED rotator controller across a VPN and fixed forty-seven differences at once (Chapter 15). It installed OpenClaw and later the Hermes platform, working through every error on its own (Chapter 13). It took a raw nine-minute IQ recording of an ISS pass and drove it through every DVB decoder available to diagnose exactly why the signal wouldn't decode (Chapter 16). In each case my job was the same: describe the goal, confirm the result, provide guidance along the way and check the results.

What This Enables

The pattern works for anything with a network interface, a serial port, or a configuration file. If it's on your LAN and Code can reach it, you can automate it by describing what you want — no need to know the CI-V byte sequences, the hamlib API, the systemd syntax, or the Python serial library. I've pasted the URL of an online project guide into Code and told it to make my Pi do what the article describes, and it did, handling the errors along the way.

If you take one thing from this chapter, let it be this: you no longer need to interact with your Pi or any other computer directly unless you want to. Give Code SSH access and describe the outcome. It is the single most time-saving capability I've found in AI, and I wish I'd realized it much sooner.

Part 3 · Learning With AI

Chapter 5 — Learning WSJT From Scratch — and Teaching It the Same Week

I had agreed to run a Tech Saturday on the WSJT digital modes before I admitted to myself that I couldn't really explain them. I used some of them on the air, but FT8, Q65, MSK144, and WSPR were four names I couldn't have told you the differences between, let alone why each one exists. So I did what this book keeps recommending: I asked Claude to teach me, starting with nothing fancier than “explain the WSJT modes to me,” and then followed every thread until I ran out of questions.

What made it click was comparing the modes side by side — by their structure and by what they're actually good at in weak-signal work. FT8 is eight-tone FSK in fifteen-second transmit/receive sequences, only about fifty hertz wide, with a Costas sync pattern and strong forward error correction; it's the general-purpose workhorse, pulling routine contacts out of the noise down around −21 dB. WSPR isn't a contact mode at all — it's a four-tone beacon a few hertz wide, sent in two-minute transmissions, built to map propagation and decodable down near −28 dB. Q65 uses sixty-five tones and is designed for the cruelest paths, EME and ionospheric scatter, where Doppler spread would smear an FT8 signal into mush. MSK144 throws short, high-rate bursts meant to catch the half-second reflection off a meteor trail on VHF. The “aha” was seeing the single pattern behind the zoo: every mode trades speed, bandwidth, and message length for sensitivity, and which trade you want is decided entirely by the path you're trying to work.

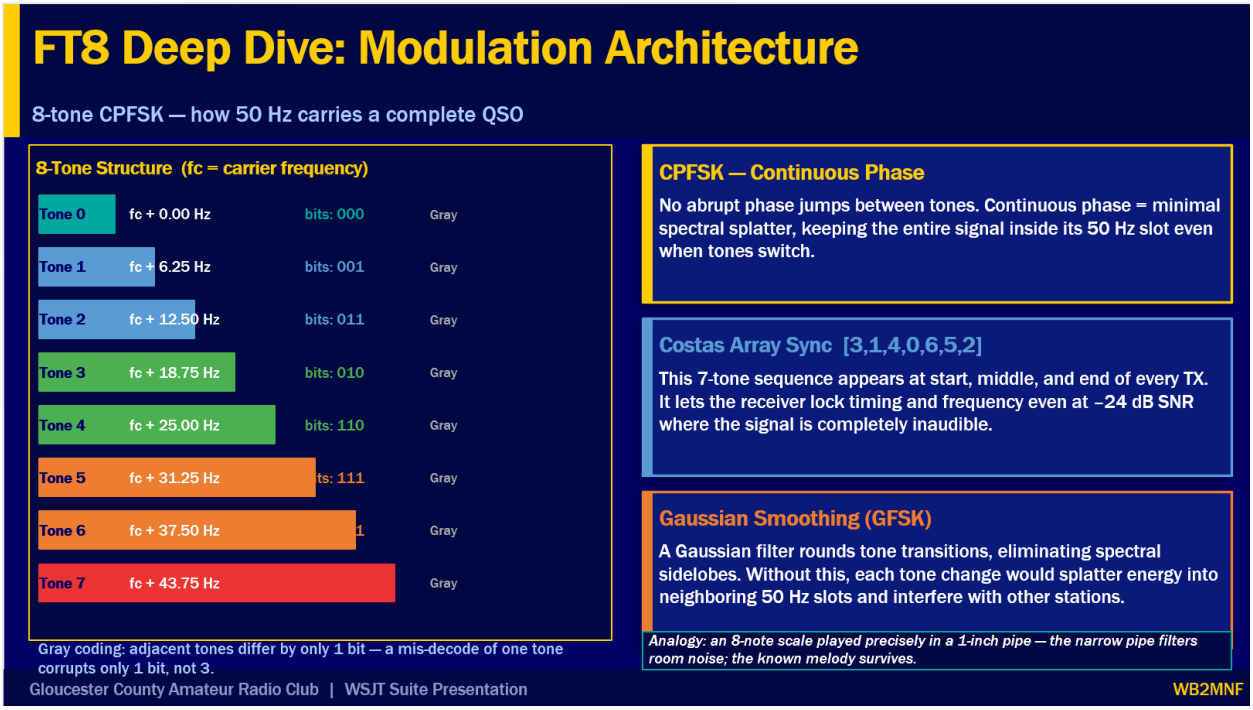


Figure 2 - PowerPoint slide on FT8

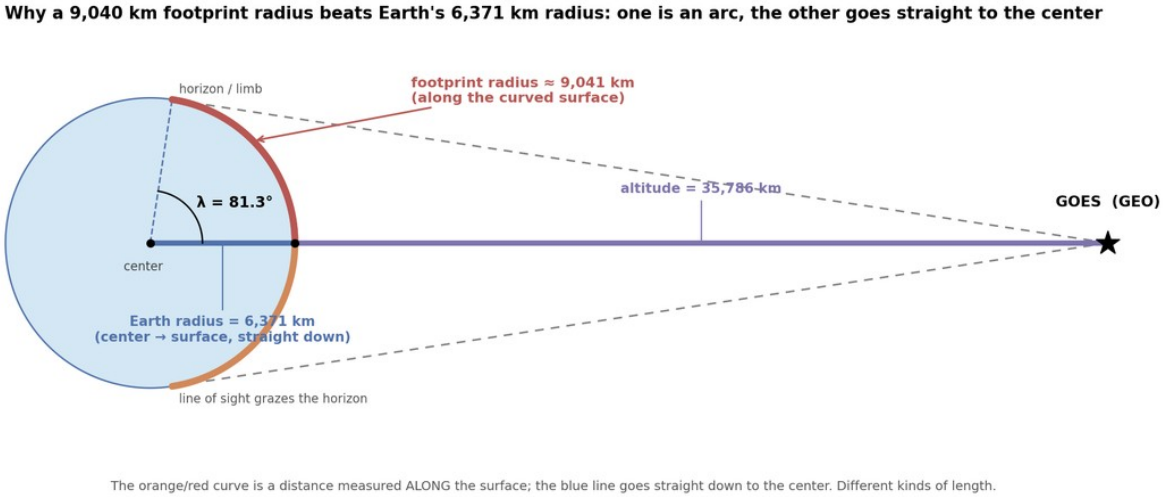
The questions kept coming — why eight tones, what the waterfall is really showing, how WSPR wrings a decode out of a signal you can't hear, what the sync arrays and the error-correction coding are doing underneath. Claude never tired of “wait, go back” or “what does that mean,” and answered each at the depth I asked for and no deeper.

Then the deliverable came out of the same conversation. I asked Claude to turn what I'd learned into a Tech Saturday presentation, and it produced a roughly twenty-slide deck — one comparison framework up front, a slide for each mode, the structure-versus-sensitivity tradeoff laid out plainly — that I was comfortable delivering to a room full of experienced operators. I had gone from “I can't really explain these” to teaching them, in an afternoon. The learning and the slides were the same session. All I supplied was curiosity and the judgment to know when an answer was solid.

Chapter 6 — Working It Out With Claude: How High Can a Satellite See?

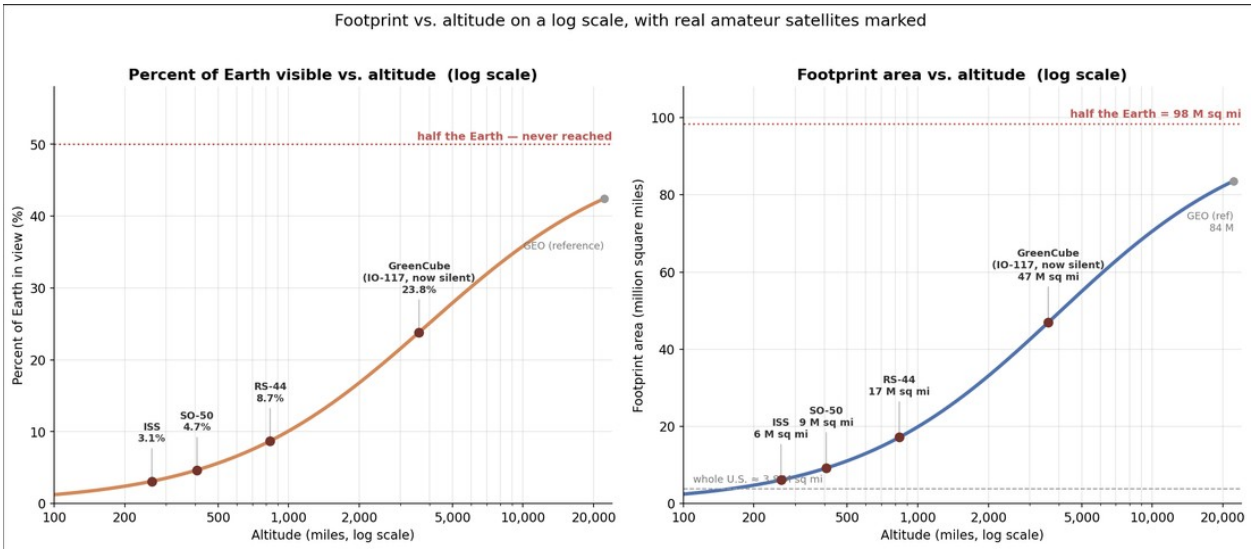
It started as a throwaway question. I wanted to know how a satellite's ground footprint related to its altitude — I figured it was a one-line answer I could drop into the space camp materials. Claude gave me the geometry and a graph, but the very first result didn't sit right: no matter how high a satellite climbs, it can never see a full half of the Earth. It only approaches a hemisphere and never arrives. My engineer's instinct said that couldn't be the whole story, and I said so. That skepticism turned out to be the engine for everything that followed — the more I pushed, the more the real physics came out. It took a whole Sunday morning.

So I pushed. When Claude told me a satellite in geostationary orbit has a footprint radius of about 9,040 kilometers, I called it out — that's larger than the Earth's own radius of 6,371, so it had to be a mistake. It wasn't. The footprint radius is an arc measured along the curved surface, not a straight line — the same way a lap around a track beats the distance across the infield. To make me believe it, Claude drew a to-scale cross-section with the three lengths side by side: the radius dropping straight to the center, the long arc riding across the surface, and the satellite sitting far out in space. Seeing them together is what actually convinced me.



I kept testing it. I threw the Moon at it as a counterexample — surely the Moon is visible from a whole hemisphere of the Earth — and instead of arguing, Claude turned the Moon into proof: we see only about 49.2% of it at any instant, a sliver short of half, for exactly the geometric reason it had been describing. It even caught me on the Sun, showing that the Sun doesn't light precisely half the Earth either — and that, because the Sun is a disk and not a point, it lights a hair more than half, something I could check for myself in the fact that daytime runs a few minutes longer than night at the equinox. I admitted I was getting it, but only slowly.

What impressed me most was how it followed me when I changed direction. I jumped to EME — our moonbounce dish throws a six-degree beam that spills well past the half-degree Moon — and asked it to run that idea backward toward a satellite. It pivoted without missing a beat and handed me the altitude at which the Earth shrinks smaller than the beam. Then I objected that most satellites just use simple whip antennas, yet a footprint can be computed from the orbital elements alone with no antenna in the math. It pivoted again, straight to the heart of it: the orbit decides where you can see the satellite, and the antenna only decides where you can hear it — and it drew a clean three-panel picture of an omni flooding the whole circle while a directional beam lights just a spot inside it. The real “aha” was simple once I reached it: you can be fairly close to an object and still see most of it — we see nearly half the Moon precisely because we're so far away compared to how small the Moon is. The whole problem collapsed into one tidy formula, the visible fraction equal to the altitude divided by twice the distance to the Earth's center — a curve that rises fast and then creeps toward half forever. “I think that's my answer,” I told it.



The last step was turning it into something a sixth-grader could read. I asked for the footprint in square miles instead of percentages, and in miles instead of kilometers. The first chart didn't work — every low-orbit satellite I actually care about was crushed together against the left edge, near the origin. I told Claude to switch the distance axis to logarithmic and to look up the real altitudes of the active satellites on the AMSAT status page, plotting a spread of them but keeping GreenCube and RS-44 no matter what. It pulled the orbital data and produced the chart that finally told the story: the ISS seeing about 6 million square miles, SO-50 about 9, RS-44 about 17, and GreenCube flung way out at 47 million — a quarter of the planet — with the whole-Earth ceiling it can never reach drawn dashed across the top. That was the one. “THAT tells the story — finally,” I wrote. A week's worth of physics, and it was the logarithmic axis that made it land in a single picture a kid could read.

Part 4A · Projects: Documents & Presentations

Chapter 7 — Getting Started: Chat-Based Projects

Every project in this book started the same way: I described what I wanted in plain English and asked AI to produce it. The projects in this chapter are the simplest version of that pattern—no code, no agents, no SSH. Just a conversation in a browser window. They move from the most common use to the most surprising: writing documents, tracking a hardware design, and building a complete web application without understanding the language it was written in.

Writing for the Club

The most common use of AI at the Skunkworks group is the most ordinary: writing. Nearly every piece of written content the group has produced in 2026 was drafted or substantially developed through AI conversation. CrossTalk newsletter articles went through iterative revision cycles—the EFHW antenna article, for example, went through three rounds of tightening in about four minutes: removing internal references, fixing terminology, sharpening the theme. EmComm capability plans spanning multiple documents were developed in extended sessions where Claude served as both drafter and devil's advocate, pushing back on assumptions and identifying gaps. The lightning protection diagram for the satellite station was produced by describing the cables and connectors in text and uploading Ron NR2B's lightning protection guide as a PDF so Claude could use the proper terminology; Claude decided on its own to produce an SVG diagram of the wall penetrations complete with protector specifications.

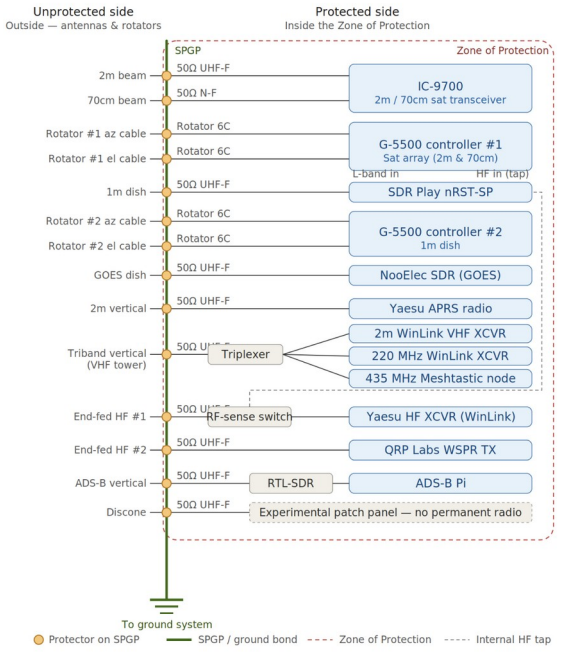


Figure 3 - SVG of Satellite Station Connections

The division of labor isn't the same for every document. For some pieces — my monthly president's report is the clearest example — I write the first draft myself, because the hard part isn't the prose, it's deciding what to talk about, and that's a judgment only I can make. Then I hand my draft to Claude to make it more readable: tighten the sentences, fix the flow, catch the repetition. For other documents the AI drafts and I edit. Either way the pattern is the same: describe what you need, get a draft on the page, then iterate. The skill is not in the writing —the AI handles that—it's in knowing what to ask for and recognizing when the result is right.

AI as a Design Secretary

A different use emerged when I worked with Chris Prioli AD2CS on his design for a 49:1 matching transformer for the club's end-fed half-wave antenna Tech Saturday project. The design was Chris's. I brought Claude in for a secondary role: keeping track of the conversation and design decisions, looking up component costs, and producing summaries and announcements for CrossTalk. I was not asking AI to review Chris's design—though, as it turned out, it did that on its own.

Chris specified an FT-240-43 ferrite core with a 3-turn primary and 21-turn secondary, wound with #14 AWG enameled wire and housed in a weatherproof polycarbonate enclosure. While tracking the design, Claude flagged that twenty-one turns of #14 AWG would not physically fit through the 1.4-inch center hole of the core. Chris challenged—he had wound toroids before and knew it would fit. I asked Claude to defend its position, and Claude did a complete reversal. It went back and calculated the wire diameter against the available area in the core's center hole, and concluded that #14 would fit with room to spare. Its initial answer had been a confident guess. When forced to do the math, it got the right answer.

This is worth noting for two reasons. First, even in a secondary role, AI will sometimes volunteer opinions that turn out to be wrong, and human expertise is the check that catches them. Chris knew the answer from experience; Claude was guessing from general principles. Second—and this is the more useful lesson—if I had been designing the transformer myself, with less toroid experience than Chris, I would have used Claude as the primary designer rather than a note-taker. In that case, the wire-fit error would have gone into the design unchallenged unless I had thought to ask Claude to go back and validate its own work. The instruction "go back and check whether that's actually true" is one of the most valuable prompts you can give, and it's easy to forget when the AI sounds confident. Don't just accept a surprising claim. Ask AI to show its work.

Designing a 3D-Printed Part

I needed a 3D-printed mounting plate to hold a WisBlock base board inside an enclosure for a Meshtastic node. I know how to use Fusion 360, but I use it infrequently enough that I'm never really proficient — every project involves

relearning commands and multiple iterations before the design is right. Instead of opening Fusion, I described the plate to Claude: the overall dimensions, the thickness, where the case mounting holes were, and the positions of the four board mounting holes measured from the edges. Claude drew a diagram first so I could verify the layout, then generated a printable STL file that I loaded directly into the Prusa slicer. It was perfect on the first print. That had never happened with any of my Fusion 360 projects.

A larger enclosure project showed the other side of that coin. The EmComm situation recorder described in Chapter 19 is a Raspberry Pi 5 with a five-inch touchscreen, and it needed a case with the Pi's ports still reachable. I asked Claude to generate the design as code rather than as a finished file. What came back was interesting less for the geometry than for how carefully it separated what it knew from what it was guessing. The Pi's board outline, mounting hole positions, and — the number that actually drives the design — the 24 millimeters of clearance needed above the board for the cooling fan were all exact. It even identified which Pi model was on the bench by reading a detail out of the machine's own startup log, since the Pi 4 and Pi 5 have different port positions.

But “eight hundred by four hundred eighty pixel HDMI screen” is sold in at least three physical sizes with completely different outlines and mounting patterns. So the design was written with the Pi dimensions baked in and the screen dimensions as clearly labeled variables, with three measurements flagged in the source as guesses and an explicit instruction not to print it yet. Telling someone “don't start this four-hour print until you've measured three things” is a more useful output than a confident file that produces scrap.

One decision from that project generalizes well beyond 3D printing: the port openings are full-width slots rather than shaped holes for each individual connector. Individual cutouts have to land within a fraction of a millimeter, and a two-millimeter error makes the whole print garbage. Slots are forgiving, take no longer to print, and sidestep the Pi 4 versus Pi 5 port-ordering difference entirely. When I asked whether the screen dimensions could be read from a photograph with a ruler in it, the honest answer was “partly” — outline yes, hole spacing risky, depth impossible — followed by a better suggestion: press a sheet of paper over the mounting holes to mark them and photograph the flat paper, which has no thickness offset and no perspective error. And a closing note that thirty seconds with calipers beats all of it.

A third print took the same approach further. The club had accumulated a stack of Heltec V3 boards — small LoRa modules with a built-in OLED display, used for Meshtastic nodes — and they needed somewhere to live other than loose on a shelf. What I wanted was a single plate that would hold four of them side by side, keep each board's antenna connector and USB port reachable, and mount to the wall of the equipment rack.

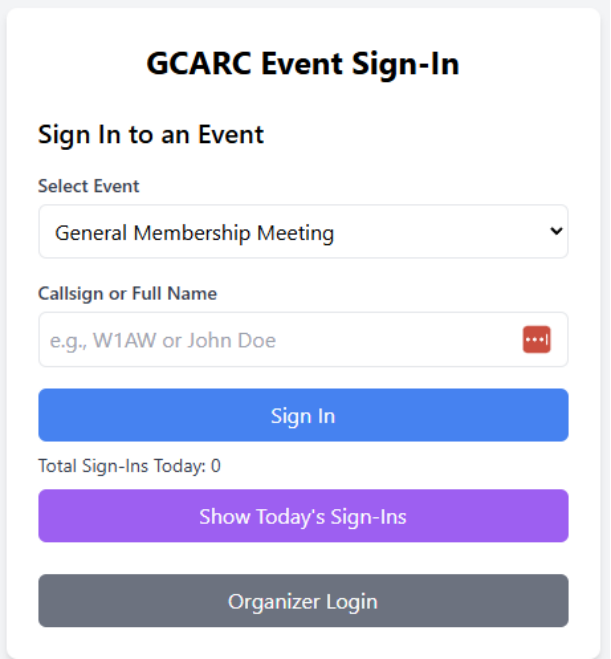
I described that to Claude in plain language: four board positions stacked vertically, the board dimensions, a retaining bar across each board to hold it in place without screws through the PCB, clearance on both sides for the antenna pigtail and the USB cable, and mounting holes at the four corners of the plate. Claude generated the model directly. The first version printed, fit, and was too thin — the plate flexed when a board was pressed into place and the retaining bars did not hold. I told it what happened, it thickened the plate and the bars, and the second print was right. Two rounds, start to finish, with no CAD software opened at any point.

The result is shown below: four positions, each with its own retaining bar and a pair of oval cutouts that clear whatever connector happens to be on that side of the board. The oval shape is the same reasoning as the full-width slots on the recorder case — a shape that is forgiving of a millimeter of error in either direction beats a shape that has to be exactly right.

Figure 4 - Hektec V3 3D Printed Holder

Building Something You Can't Build Yourself

I cannot write HTML. I only have a basic understanding of JavaScript. I have no training in web development and no interest in acquiring any. Despite this, the GCARC event registration system—a live web application that the club uses to track attendance at every function—exists, works, and has been in production for months. I built it by talking to Grok.



The application lives at gcarc-event-registration.web.app. Under the hood it is 862 lines of HTML and JavaScript backed by a Firebase database with Google authentication, three data collections for events, members, and individual sign-in records, and a reporting system that can filter by event type and date range and export the results to CSV. It has an organizer login, an event creation interface, a member roster manager, and a public sign-in screen that members use on a tablet at the door. It is, by any reasonable measure, a real piece of software. I did not write a single line of it.

The process was pure vibe coding. I told Grok what the sign-in screen should look like. Grok wrote it. I said I needed an organizer section behind a login. Grok added it. I wanted a way to create new event types, add members to the callsign table, and generate reports filtered by date. Feature by feature, screen by screen, I described what I wanted and Grok produced the code. When something didn't work the way I expected, I described the problem and Grok fixed it. Grok also taught me how to set up the Firebase database and how to use the Firebase console to manage it—skills I needed exactly once and would never have learned from documentation.

What Grew Behind It

The registration system turned out to be more valuable for what it enabled than for what it did at the door. Once every sign-in was recorded in a database, questions that had been unanswerable became trivial: which members attend regularly, which activities draw the most participation, and—critically for a club trying to understand why some members don't renew—which members stopped showing up and when.

That required additional code, most of it also AI-written. A utility script—mostly Grok, partly Claude—takes the monthly member list produced by Jeff WB2ZBN and updates the member database automatically. When we held the 2026 club picnic, I had AI write custom code to import the participants directly from the picnic registration list into the system as attendees of a new activity, rather than entering fifty names by hand. Claude wrote the code that runs on my production agent to produce monthly participation reports from this data, so the board can see at a glance how the club's activities are performing.

None of this required me to understand the underlying code. I described what the data should look like going in, what the report should look like coming out, and AI handled everything in between.

The Starting Point

These are the projects I'd recommend to anyone trying AI for the first time. They require nothing more than a browser and a question. You don't need a Raspberry Pi, an SSH connection, or a programming language. You need a task—a document to write, a conversation to track, a system to build—and the willingness to describe it out loud. The chapters that follow build on this foundation with progressively more powerful tools. But every project in this book, no matter how technical it eventually became, started right here: a person typing a description of what they wanted into a text box and seeing what came back.

Chapter 8 — Automating the Monthly Meeting Deck

Every month the GCARC general membership meeting opens with a PowerPoint deck—meeting date, new-member welcomes, the Tech Saturday topic, the month's featured event, a calendar of activities, the upcoming-programs schedule, and Tony Starr K3TS's always-popular contest picks. Building it by hand means re-reading the newsletter, recalculating a half-dozen dates, hunting through email, and editing the same slides the same way every time. It's the kind of work that's easy to describe but tedious to do, and it has to happen twelve times a year without fail.

I handed the tedious part to Claude.

What Goes In

The update draws on three inputs. The first is last month's deck, which serves as the template to roll forward—most slides stay the same; only the dates, names, and details change. The second is the current month's CrossTalk, the club's newsletter, a 60-page PDF that supplies the new-member roster, the Tech Saturday topic, the featured event details, and the confirmed program schedule. The third is Tony's contest picks email, pasted in directly, which provides the verbatim text for the contest slide.

Underpinning all of it is a runbook—a document I wrote once that describes, slide by slide, what changes each month, how the meeting dates are calculated, and which slides never change. The runbook is the AI's instruction manual. I give it the three inputs and the runbook, and it knows what to do.

What Claude Does

Claude starts by computing the month's dates from a single rule: the meeting is always the first Wednesday, and everything else—Tech Saturday, license testing, the directors' meeting, the clubhouse dinner—follows from it. For June that fixed the meeting on the 3rd and Field Day on the weekend of the 27th–28th.

It then reads the CrossTalk PDF and pulls out what it needs: the new members with their callsigns, license classes, and towns; the Tech Saturday subject and presenter; the featured event details; and the program lineup for coming months. It reads Tony's email and formats the contest picks with gold headers to match the deck's house style.

Then it opens the deck and edits each target slide. During the June update, it discovered that the slides had drifted out of the positions the runbook specified—someone had added or moved slides at some point. Rather than blindly editing slide number 9 and getting the wrong one, Claude located each target slide by reading its content. It updated the title date, the new-member list, the Tech

Saturday slide, the featured event, the events calendar, the programs table, and the contest picks.

Catching Its Own Mistakes

After editing, Claude rendered each changed slide to an image and inspected it visually. That self-review caught a real problem: Tony's June contest text was longer than usual and overflowed the bottom of the slide. Claude reduced the font size and tightened the line spacing, rendered it again, and confirmed everything fit. It also checked every edited slide for stray references to the previous month's date—the kind of leftover that's easy to miss when editing by hand and embarrassing to display in front of fifty members.

What I Do

It's important to be clear about the division of labor here, because it's narrower than "Claude builds my deck." I build all the slides that carry the actual content of the meeting — the substance I'm there to present. Claude doesn't create any of that. What Claude does is mechanical: it brings in content that already exists elsewhere — dates computed from a rule, names lifted from the newsletter, contest text pasted from Tony's email — and drops each piece onto the right slide in the right format. It's assembling and transcribing, not authoring. My role each month is to supply the files, build or update the content slides, answer the occasional clarifying question (no license-prep session was announced this month), and give the final deck a look. Claude also flags judgment calls for me—the treasurer's year-to-date figures were left untouched because the newsletter didn't break them down the way the slide requires, and the runbook's slide numbers needed refreshing to match the current deck. Those are decisions for the human. The reading, calculating, transcribing, formatting, and proofing are the parts worth handing off.

Why This Matters Beyond the Club President

Only one person in the club needs to update this particular deck. But the pattern applies to anyone who maintains a recurring document that pulls from multiple sources on a regular schedule. A net control station report. A repeater trustee's monthly status update. A contest committee's post-event summary. Any time you find yourself doing the same mechanical editing work month after month, reading the same kinds of sources and dropping their content into the same slots, you have a candidate for this approach: write a runbook once that describes what changes and where the information comes from, then let AI do the mechanical work each cycle.

The runbook is the key. Without it, you'd have to re-explain the process every month. With it, the AI has standing instructions that carry forward indefinitely. Write the runbook once. Use it forever.

Chapter 9 — Building the Ham Space Camp Curriculum

The Space Camp that the GCARC produces in cooperation with the Upper Deerfield School District is six sessions that carry a student from radio waves to hearing a live satellite pass, and the CubeSat Simulator experiments that run alongside them. This chapter is not about that content. It is about the documents: how the raw material for the camp became a package a volunteer could actually pick up and teach, and how an AI helped assemble that package and then, at the right moment, tear it apart and put it back together better than I had it.

The camp itself is six sessions, but building the teaching package behind it was not a week's work — it was months of effort and, honestly, hundreds of hours of pulling materials together, spread across dozens of working sessions, several of which filled Claude's context window and required starting fresh. This was a sustained partnership: I supplied the domain expertise and pedagogical judgment, the AI supplied the drafting, document engineering, research, consistency-checking, and the patience to rebuild documents from scratch as many times as the material required.

The Package

What the camp needed was never one document. It was fourteen deliverables that had to agree with each other: six session decks, a sixty-seven-page Student Workbook, a forty-six-page Instructor Guide with scripted language and anticipated student questions, eight CubeSat Simulator experiments adapted from Dr. Alan Johnston KU2Y's AMSAT activity guides, a calculator skills manual, hands-on lab worksheets, career spotlights, pre-camp film discussion questions, a family radio research guide for parents, custom cover art, and a running Pending Changes Log that tracked every approved edit. The log turned out to be essential—changes were batched and applied in clean rebuilds rather than patched in piecemeal, which kept documents internally consistent as the content evolved.

The piece I would not have thought to build on my own was a master visualization inventory cataloguing every diagram, slide, and figure across all six sessions. That inventory became the spine of the effort. Once every visual lived in a single list, it was possible to see where an idea was explained twice, where a figure was missing, and where the sequence jumped.

The Calculator Manual

The students are each receiving a TI-84 graphing calculator—new territory for most middle schoolers. Claude wrote a skills manual covering the specific calculations they'd perform during camp, exploratory bonus problems, and a guide to which calculator functions to ignore. This document was a vivid lesson in how a late change ripples through a package: we had originally planned on a Casio and switched to the TI-84 at the last minute, which forced a complete rewrite of the

calculator manual and every place in the curriculum that referenced a keystroke or a screen. The manual then went through several more rebuilds as we found errors. At one point the keystroke sequence Claude provided for a particular function simply didn't work on the actual TI-84. I caught it because I checked, we researched the real interface, and every occurrence of the incorrect sequence across every document was corrected. The pattern: the AI drafts at speed, the human verifies against reality, and when an error is found it gets fixed everywhere.

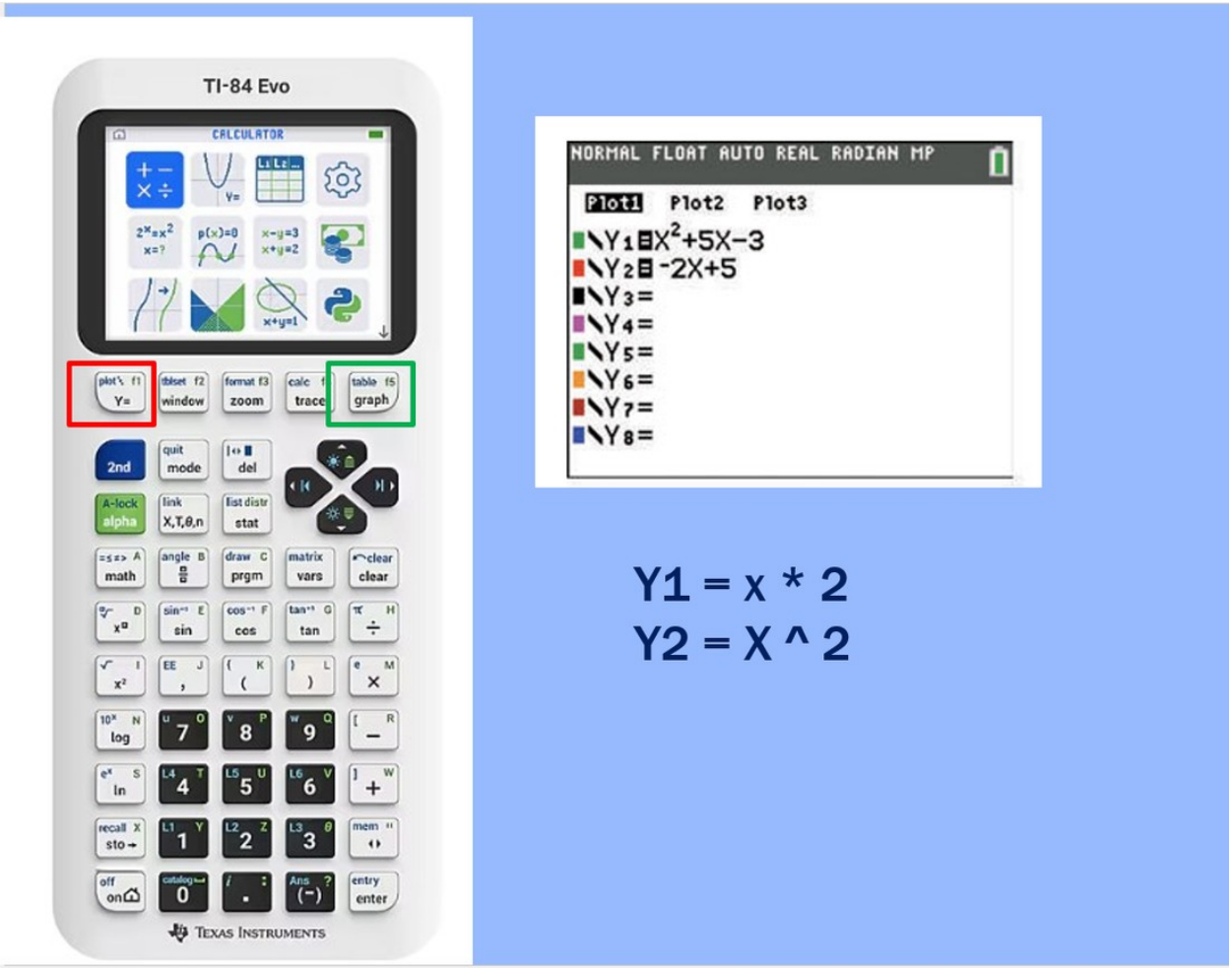


Figure 5 - PPT Slide on Formula Graphing

How Revision Worked

Revision happened at three scales. At the sentence level, passages were reworked until they were accurate and the right weight for a young audience. At the document level, the rule was to rebuild rather than patch—a dozen small in-place edits leave a session subtly incoherent, so when a session needed real work we regenerated it whole. At the structural level, two habits mattered: cutting rather than simplifying when something was too advanced, and killing redundancy by cross-reference rather than repetition so each idea lived in exactly one place.

The Whole-Package Pass

The turning point came when the draft was complete enough to stop adding and start reading. I had Claude review the entire package end to end—not one session, all of them—and the value was something no amount of slide-by-slide editing could have produced. Read as a whole, the material's problems were problems of order, not content. Modulation modes introduced piecemeal across several sessions were consolidated into one. An analogy buried in a technical aside got moved earlier, where it actually did some teaching. Explanations that had drifted apart between session one and session five were reconciled against a single definition. None of this is visible when you edit one screen at a time. It only surfaces when something holds the entire package in view at once—precisely the task a multi-file project is worst at by hand and best at with a partner that does not lose the thread.

Staying the Editor

This only worked because I kept treating the AI as a fast collaborator and not an oracle. It was confidently wrong on occasion—the calculator navigation, a coax loss figure that needed checking, a satellite parameter that had changed. In each case, the correction was straightforward once caught. The AI researched the real answer, and every instance across every document was updated. That is the right division of labor: the model holds the package together and proposes the restructuring, and the human stays the editor, the fact-checker, and the final authority. For material that will be taught to children, that authority is not optional.

The result was fourteen documents totaling well over a hundred pages, internally consistent, correctly sequenced, and ready for a volunteer to teach. The camp starts July 6, 2026 at the W2MMD clubhouse with seven students selected by Angela KE2DRJ. More usefully for any reader, the project demonstrated a repeatable method: keep a single authoritative view of the whole, maintain a change log, rebuild rather than patch, and at the end let the AI read everything at once and tell you what order it should have been in all along.

Chapter 10 — Teaching AI to Teach: The FCC License Trainer

I wanted to build a free AI tutor that could help someone prepare for the FCC Technician class license exam. It needed to do two things: teach the candidate the material behind each test question, and then administer a realistic practice test. The practice test had one absolute rule: when the candidate picks the wrong answer, the tutor must explain why that answer is wrong without revealing which answer is correct. The learning happens in the struggle to figure it out, not in being handed the answer.

Grok couldn't do it. Every wrong answer got a response like "No, A is wrong, the correct answer is C." Claude handled it on the first try and has never violated the rule across hundreds of test questions. The system is live at skunkworks.w2mmd.org/license/ and works well on Claude. But Claude requires a subscription. I wanted this to run on a free model so any prospective ham could use it without paying.

The Setup

The system runs on my home equipment—the Mac Mini serving the AI model, a web-based chat interface that students access through a browser, and a database loaded with all 410 questions from the official FCC question pool. A Cloudflare tunnel makes it accessible from outside my network at ai.wb2mnfai.org, so a student anywhere can reach it without my opening any firewall ports.

The challenge was finding a free model smart enough to follow the no-reveal rule reliably. Claude Code evaluated five different free models, testing each one against the quiz requirements. The best candidate was a capable free model—but, as we discovered, not quite capable enough.

What Claude Code Did

This is the core of the story: I asked Claude Code to make the free model work as a license test tutor. What Code did over the next several hours was something I could never have done manually—it built an automated testing system, ran the free model through it hundreds of times, identified exactly how and when the model failed, attempted to fix each failure, and tested again.

Code wrote Python scripts that simulated students taking the test—answering questions right and wrong in various patterns—and automatically detected five specific types of failure. Did the model reveal the correct answer when the student got it wrong? Did it freeze for minutes before responding? Did it fail to find the right question in the database? Did it mark a wrong answer as correct? Did it describe the answer choices before the student had a chance to pick one? The scripts caught all of these automatically, every time, across dozens of test scenarios per run.

When failures were detected, Code would analyze the pattern, write a more specific instruction to the model addressing that exact failure, add it to the model's instruction set, and run the tests again. Roughly a dozen versions of the instruction set went through this loop. Some instructions helped. Some helped with one problem but caused a different one. Code tracked all of it.

The Surprises

The most important discovery was that some of the failures weren't the model's fault at all. They were bugs in the surrounding software.

The biggest one: when a student typed just a single letter as their answer—"B"—the system that looked up questions in the database was searching for the literal letter B instead of the actual test question. It would return a random, unrelated question, and the model would respond to the wrong question entirely. Code found the bug, rewrote the search logic for this specific use case, and the question-retrieval failure rate dropped from fifty percent to zero. That one fix did more than all of the instruction-set changes combined.

Another discovery: the model was spending one to two minutes "thinking" silently before each response—working through its reasoning internally before producing visible output. Code found a way to suppress this behavior, and response time dropped from nearly two minutes to seven to twenty-five seconds. The student experience went from unbearably slow to conversational.

The Results

After fixing the software bugs, the system improved dramatically. The baseline had 22 failures across 63 test scenarios. After the model upgrade, that dropped to 2. After the software fixes, it dropped to zero.

Then Code over-iterated. Adding more and more rules to the instruction set started causing new problems—the rules competed with each other, and the model would follow one rule at the expense of violating another. The instruction set had grown to 24 kilobytes of text. Code deliberately cut it back to 7 kilobytes, and performance improved. More instructions did not mean better behavior. Past a certain point, it meant worse.

The Honest Conclusion

The system is deployed and useful, but it's not reliable enough to trust unsupervised. After more than a dozen rounds of testing and refinement, the remaining problems—occasionally revealing the correct answer, sometimes losing track of where it is in a multi-question sequence, intermittently ignoring its own rules—are at the limit of what this generation of free models can do. Further instruction changes show diminishing returns: fixing one problem tends to cause another.

The project is on hold, waiting for smarter free models. The entire setup—the Mac Mini, the web interface, the question database, the testing scripts—is ready to go. When a more capable free model appears, and they improve rapidly, it will likely require nothing more than swapping in the new model and running the same tests. The four things to check: can it maintain context across four or more questions without losing its place, can it consistently withhold the correct answer, can it handle "all of the above" questions correctly, and can it teach one question at a time without racing ahead.

What This Chapter Is Really About

The license trainer is the most technically involved project in this book, but the story underneath the technical details is simple. I asked Claude Code to make a less capable AI model perform a task it wasn't quite smart enough for. Code responded by building an automated testing system that ran more tests, more carefully, and more patiently than any human could. It tested the model hundreds of times, rewrote instructions dozens of times, discovered bugs in the surrounding software that I would never have found manually, and at the end gave an honest assessment: this model isn't ready yet, but the infrastructure is, and the next one might be.

One AI training another AI, with more patience and rigor than a human could sustain. That's the chapter.

Part 4B · Projects: Operational Assistance

Chapter 11 — DMR Codeplug Translation

Every DMR radio stores its programming in a codeplug—a data file containing channels, contacts, talkgroups, zones, scan lists, and dozens of vendor-specific settings. When you buy a new radio, that codeplug needs to be rebuilt from scratch in the new radio's programming software, because every manufacturer uses a different file format with different field names, different value conventions, and different import requirements. Transferring a working codeplug from an AnyTone to a Retevis, or from a Retevis to an OpenGD77 radio, is traditionally an afternoon of tedious manual data entry. Or you can let AI do it in minutes.

The Three-File Method

The process is the same regardless of which radios are involved. You need three things: the source data exported from your current radio, a template showing the target radio's expected format, and an AI to map between them.

First, export the codeplug from your current radio's programming software to CSV. This gives you the complete channel list, contact list, and other data in a text format that AI can read.

Second—and this is the step that makes the whole thing work—open the new radio's programming software, manually enter one or two channels with real data, and export that file. This sample export serves as a Rosetta Stone: it shows the AI exactly what field names the target software expects, what format the values should be in, and what valid entries look like. For example, the sample tells the AI that the channel mode field contains either "Analog" or "Digital" rather than a number code, or that the power field uses "High" and "Low" rather than wattage numbers. Without this sample, the AI would have to guess at format conventions, and the target software would reject the import.

Third, give both files to the AI and ask it to translate the source data into the target format. The AI reads both CSVs, maps the fields, and produces a new CSV that the target software will accept.

AnyTone to Retevis: What the Translation Looked Like

The first time I did this was translating my AnyTone AT-D878UV III codeplug to a Retevis Ailunce H1. The AnyTone codeplug contained 200,000 contacts in the master list and 365 programmed channels. The H1's programming software had completely different field names, different column ordering, and strict import validation that rejected files with even a one-word difference in a column header.

The translation required several iterations. The AI initially copied identical receive and transmit frequencies for hotspot channels, assuming simplex, when they should have been duplex. It set timeout values incorrectly on two channels. Some of the AnyTone's longer channel names were truncated by the H1's shorter display,

so the AI shortened them to readable abbreviations. Each problem was caught by attempting the import, reading the error message, and feeding it back to the AI for correction. The final result: a 74-channel codeplug with correct frequencies, offsets, talkgroups, and permissions, plus a 48-channel roaming list built by filtering and deduplicating the AnyTone's 365 channels. No manual data entry.

The Same Pattern, Any Radio

The method works for any two radios for which you have the programming software and can export and import CSV files. The OpenGD77 firmware uses yet another format. The AnyTone uses yet another. The field names are different, the column ordering is different, the value conventions are different. The three-file approach handles all of them: export the source, create a sample in the target, give both to AI.

Building a Contact List from a Newsletter Screenshot

A separate but related project: the GCARC CrossTalk newsletter publishes a table of club members' DMR IDs every month. I wanted those contacts in my radio for private calling without typing fifty-plus entries by hand.

I copied a screenshot of the CrossTalk table and pasted it directly into Grok. Grok read the image, extracted every callsign and DMR ID from the table, and produced a CSV formatted for direct import into the radio's programming software. No retyping, no transcription errors, no missed entries. The CSV went straight into the codeplug as private calling contacts. The entire process—from screenshot to imported contacts—took less time than it would have taken to type ten entries by hand.

This generalizes beyond codeplugs. Any structured data that exists in one format and needs to exist in another—a membership list becoming a contact import, a spreadsheet becoming a configuration file, a PDF table becoming a CSV—is something AI handles naturally. You show it the source, show it the target format, and ask it to translate.

Chapter 12 — The WSPR Network

The GCARC WSPR Network is a club-wide crowdsourced propagation study. Members build TAPR WSPR transmitter boards, pair them with Raspberry Pi 3s, and deploy them at home to transmit weak-signal beacons on multiple HF bands. The spots are collected by WSPRnet and displayed on the club's propagation dashboard at skunkworks.w2mmd.org. Over two Tech Saturday build sessions, roughly eighteen participants assembled boards, including several members from the South Jersey Radio Association who joined us for the project.

The hardware was straightforward. The software was not.

The SD Card Problem

Every WSPR beacon needs a Raspberry Pi with the right software, the member's callsign and grid square, and a way to get online so the Pi can pick up a timing signal to synchronize its transmissions. The alternative to pre-programmed SD cards was having each member sit down at a computer during the build session to configure their own Pi—downloading software, typing Linux commands, entering their callsign and coordinates. With eighteen participants whose experience with Raspberry Pis ranged from expert to "what's a Pi," that would have consumed the entire session. We needed a card that any member could insert, power on, and forget about.

An earlier attempt to build a standard image had failed—the tool we used to shrink the image file corrupted the boot process on some Pi 3s, producing a stack of cards that wouldn't start. We had to rebuild from scratch.

What Claude Code Built

Working on Bob—my test Pi (described in Chapter 3)—Claude Code built the entire SD card system over several long sessions. It started from a stock Raspberry Pi OS image, removed the desktop software to save space, and installed the WSPR-specific pieces: the transmitter software, a web-based control panel accessible from any browser on the home network, and a WiFi setup system that would guide a non-technical user through connecting the Pi to their home network.

Getting members onto their home WiFi was the hardest problem. The solution is a three-stage flow. First, if an Ethernet cable is plugged in, the Pi uses it and starts transmitting—no WiFi needed. At the build sessions, we ran an Ethernet cable from the clubhouse router so members could verify their beacons were working immediately, without dealing with WiFi setup at all. That saved significant time. Second, if the Pi remembers a saved WiFi network from a previous setup, it connects automatically. Third—if neither works—the Pi creates its own temporary WiFi hotspot called WSPR-Setup. The member connects their phone to that

hotspot, and a setup page pops up asking for their home network name and password.

Before committing to the home network, the Pi does something that eliminated most of our early support questions: it briefly tests the credentials, captures the IP address the home router assigned, and shows the member a confirmation page with that address and a scannable QR code pointing to the control panel. Only after the member taps "Finish setup" does the Pi switch to the home WiFi permanently. The member knows exactly how to reach their beacon from any device on their network.

Burning Cards for Members

For each participant, the master image gets written to a fresh SD card with the member's callsign, grid square, band, and transmit power patched into a configuration file. Code took the registration list and automated the process: it worked through the list of registrants one by one, prompting me to insert a blank card into Bob's SD slot, then wrote the image, patched in that member's callsign and band, and told me when to swap in the next card. A fast card burned in about nine minutes; the slower recycled cards took closer to twenty. The first batch produced 24 cards.

With the cards burned at home, we carried the whole stack to the clubhouse for the Tech Saturday work session. That's where the build actually came together: members assembled their TAPR boards, paired each with a pre-burned card, and we tested every device on the spot. With an antenna connected, we checked wspr.net for spots from distant stations to prove each beacon was functional. All but one produced spots—the exception was a 10-meter beacon tested on a day when 10-meter propagation was nonexistent, which is a band-conditions problem, not a device problem.

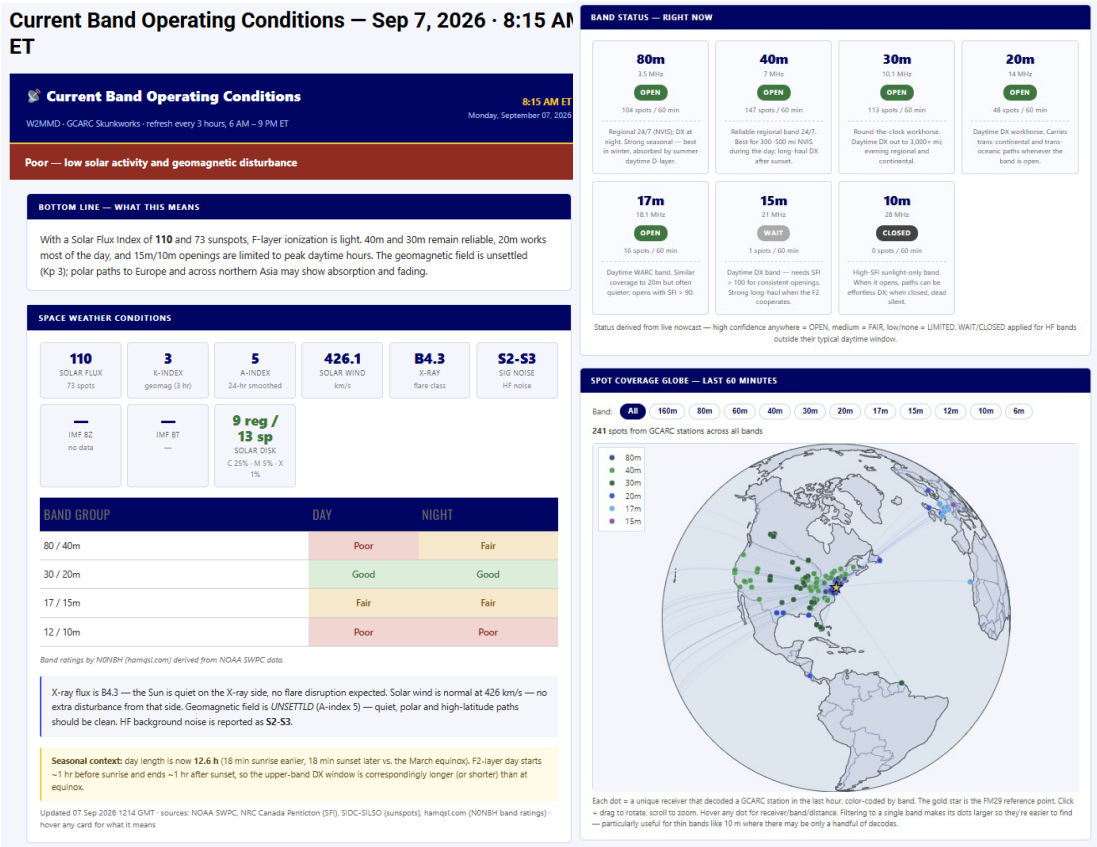
After member feedback identified two issues—the setup page occasionally blocking the control panel after setup was complete, and the WiFi radio on some Pi 3s coming up in a disabled state at boot—Code rebuilt the master image with fixes for both. And despite all of this development, one member still couldn't get their beacon onto their home WiFi. Members' experience with Pis varied widely and we had to accommodate everyone, which sometimes meant a house call. That's the reality of deploying technology to a diverse club membership.

Automated Propagation Reports

The other half of the WSPR Network is what happens with the data. The AI agent described in Chapter 3—the one that runs continuously on my home equipment—pulls data every three hours from two sources: NOAA's Space Weather Prediction Center for solar flux and geomagnetic conditions, and WSPRnet for actual spots received from the club's beacons and others worldwide. It feeds both datasets to a language model with instructions to interpret the data as a propagation

observatory—reporting what the bands are actually doing based on measurements, not forecasting what might happen.

The model generates a plain-English band conditions summary and the agent publishes it to the club website automatically. The report appears at wspr.wb2mnfai.org. The agent also handles the AI interpretation layer on the WSPR participant dashboard. None of this requires my involvement. I set it up, tested it, and it's been running autonomously ever since—six reports a day, seven days a week.



The 1500 Hz Offset Bug

One technical problem uncovered during the project is worth mentioning, not because it affected many members—the QRP Labs beacon is only used at the clubhouse and by one other individual member—but because Claude's explanation of it serves as a partial tutorial on how WSPR works. The QRP Labs beacons on 20, 15, and 10 meters were transmitting at dial frequency plus 300 Hz instead of the correct dial plus 1500 Hz. The fix was to set the configured frequency 1200 Hz higher than the published WSPR dial frequency for those bands—a workaround for a firmware behavior that the QRP Labs documentation didn't clearly explain. Claude produced a full HTML article with an SVG frequency diagram for the Skunkworks website that walked members through both the problem and the underlying WSPR frequency conventions.

Satellite Passes and Weather Pages

Claude Code also built live pages on the Skunkworks website supporting other club activities. A Saturday Satellite Passes page pulls current orbital data, computes upcoming passes for the W2MMD location, and displays pass times, elevations, and frequencies—auto-refreshing so the information is current when members check Saturday morning. A weather page shows conditions relevant to outdoor operating. Student-formatted versions of both pages were produced for the STEM class, with simplified displays and plain-language summaries. All written, scheduled, and deployed by Code.

Three Roles for AI

The WSPR project used AI in three distinct ways. Claude Code was the builder: it wrote the SD card system, the WiFi setup flow, the burn script, and the website pages. The AI agent was the operator: it runs the propagation reports autonomously on a schedule, without human involvement. And Claude in chat was the explainer: it wrote the offset-bug article and the member-facing documentation that made the network usable by people who weren't part of the build team. Builder, operator, explainer—three roles, one project, all AI.

Chapter 13 — Mesh Networking: When Nothing Announces Itself as Broken

The club runs mesh networking nodes on LoRa — a low-power, long-range radio technology that lets small battery-operated devices relay text messages across a town without any infrastructure. It is genuinely useful for emergency communications, and it is also, in my experience, the single most frustrating thing in the hobby to get working. Not because the concepts are hard, but because almost every failure mode looks exactly like success.

A node boots. It answers every command. Its status page reports healthy. And it radiates nothing. Another node has a channel configuration that matches its neighbor byte for byte, and hears nothing. A third accepts the setting you give it, reports back a different value, and does not flag an error. This chapter is about three mesh projects and the discipline that AI assistance made practical: verify everything, trust nothing that has not been read back.

Meshtastic: A Node You Cannot Reach

The clubhouse tower at the Gloucester County 4-H Fairgrounds is 89 feet tall. We wanted a one-watt Meshtastic node at the top of it, replacing a 100-milliwatt unit, running on solar. The hardware decision was easy. The constraint that shaped the entire project was that once the node is up there, nobody is climbing the tower to reconfigure it.

That meant remote administration had to work before the node went up, and remote administration in current Meshtastic firmware uses public-key cryptography rather than the older admin-channel method that most online tutorials still describe. There are exactly three administrator slots on a node. All three are now assigned: my home station node, a portable node, and a node permanently attached to Bob. Adding a fourth would mean evicting one of them. Bob can now query and reconfigure the tower node over the air, verified end to end by reading back its transmit power setting from the ground.

Getting there took longer than it should have, and every delay came from the same category of problem. The configuration tool has two commands that look interchangeable but are not: one replaces the entire list of authorized administrators while the other only appends to it, and there is no obvious way to clear a bad entry. The node reports two different public keys in two different places, and the one displayed in the node list can be stale — authorize the stale one and you get an administrator relationship that looks perfectly configured and cannot actually talk to anything. The firmware also silently clamps values outside its accepted range: a telemetry interval set to 900 seconds read back as 1800, with no error message. And two nodes with byte-identical channel settings, one of them

completely deaf — the channel matched, but the radio preset did not, which changes both the data rate and the frequency slot.

The AI Got It Wrong, and the Human Caught It

Asked to determine the club's existing mesh configuration, Claude examined a nearby node, concluded the mesh was running on 433 MHz with a long-range slow preset, and configured two nodes to match. It was wrong. That node belonged to a completely separate network. The correct configuration was 915 MHz with a long-range fast preset.

I caught it because I knew which node actually worked in the field — one that had heard about thirty other nodes while being driven around the county. The standing rule since then is that the canonical reference is the node known to work in the field, not whichever node happens to be convenient to query. This is the same pattern as the toroid design in Chapter 7: AI produces a confident, plausible, wrong answer, and human domain knowledge is the check that catches it.

Programming the Nodes With Claude Code

Something that is not obvious until you try it: Meshtastic and MeshCore nodes are configured from a command line, and Claude Code can drive a command line. That means Code can program your mesh nodes for you. You describe the configuration you want in plain English — set the tower node to long-range fast, 915 MHz, full power, power saving off — and Code issues each command, reads back every value, and tells you when the radio accepted something different from what you asked for. Given how much of this firmware silently clamps or ignores settings, the read-back is not a nicety. It is the entire point.

MeshMon: Watching Without Interfering

Once the tower node was up, we needed to know how it was actually performing. I asked Claude Code to install MeshMonitor on Bob. That was the whole request. Code discovered that MeshMonitor runs inside Docker, that Docker was not installed, and that it would therefore need to install Docker first. It added the software repository, installed the container engine and its supporting packages, set the permissions, started the service, configured it to launch at boot, pulled the MeshMonitor image, wrote its configuration, brought it up, and verified the dashboard was reachable. I never typed a command, never looked up what Docker needs, and never knew there was a dependency to solve.

The critical design decision was that the monitor is entirely passive. It only listens. It never transmits. That matters because it adds zero airtime and zero battery load to a solar-powered node that was simultaneously being run at worst case — full power, power saving disabled — for a drain test. A monitoring system that interferes with the thing it monitors is worse than no monitoring at all.

Three findings came out of it that a reader can use directly. The link between the tower and my home station is asymmetric: the tower hears my station roughly nine decibels better than my station hears the tower. Same pair of radios, same air. That is an antenna problem, not a protocol problem. A battery level reading of 101 percent is not a bad reading — it is a sentinel value meaning the node is on external power, where the percentage is meaningless but the voltage is genuine. And packets from the locally-attached node carry no signal-quality data and will swamp any airtime analysis unless you exclude them; only over-the-air traffic counts.

A twice-daily health report is generated by a language model running locally on the Mac Mini — no cloud, no API charges. Its first run caught the link asymmetry unprompted, before anyone had gone looking for it. One problem remains open: the fixed home node hears only the tower node, even after its configuration was matched exactly against the known-good reference. That points at range, siting, or a firmware difference in how the radio calculates its frequency slot. I do not know the answer yet.

MeshCore: The Same Hardware, a Different Network

MeshCore looks like Meshtastic from across the bench. The same boards, the same radio module, the same USB cable. It is a completely different protocol. The Meshtastic command-line tool cannot see a MeshCore node at all, and the monitoring system described above is blind to it. One radio serves one protocol. Running both networks means owning two radios. That single sentence would have saved us a weekend.

The best failure in this entire chapter came from flashing firmware. These radio boards come in variants that look identical and are not — they use different internal connections and require a specific control pin to key the power amplifier. Flash the wrong-but-plausible variant and the board boots normally, answers every command on its console, reports itself healthy, and emits no usable radio signal. Nothing tells you. There is no error, no warning, no indication of any kind. This is the cleanest example I have of the failure class that dominates this work.

The second-best is mechanical. The board has two antenna connectors — one for the 2.4 GHz Bluetooth radio and one for the 915 MHz one-watt module — and the two plugs will physically seat in either socket. Cross them and you put a full watt into an antenna designed for a completely different band, which destroys the amplifier. Related: always connect the antenna before applying power. Claude also had to correct its own earlier advice here. It initially suggested raising the transmit power setting to 30. That is wrong for this board: the radio chip's own ceiling is 22, and the external amplifier takes that to roughly 30 at the antenna. Setting 22 already means running a watt.

Most instructive of all: the clubhouse repeater had been effectively invisible for months. Its clock was set to May 2024, its advertisement interval was zero, and its

flood-advertisement interval was 47 hours. The radio was transmitting perfectly and was simply undiscoverable.

LoRa APRS: When the Answer Is “Not That Way”

A group of members wanted to repurpose old 433 MHz LoRa boards from Meshtastic to APRS — relaying position reports rather than text messages. They supplied an article and its associated code repository and asked for a written guide to flashing and configuring the devices.

What came back was that the suggested repository was the wrong target, for three independent reasons, each verified against the actual source code rather than the article describing it. First, it has no support for the board named in its own title — the build configuration defines five targets, all of them different hardware, and not one file in the project mentions the board in the repository name. Second, it is abandoned: an unmodified copy of someone else's work, last touched in November 2023. Third, it is the wrong application entirely — it is tracker firmware that beacons its own position, while the club asked for a digipeater that relays other stations. The right target was an actively maintained project with genuine support for the board, a browser-based flashing tool requiring no development software, and a single firmware image that can act as a digipeater, an internet gateway, or both.

One finding is not a technical problem at all. FCC rules cap unspecified digital codes on 70 centimeters at 100 kHz of bandwidth. The worldwide LoRa APRS standard is 125 kHz. Three defensible options were laid out — narrow the bandwidth and lose interoperability with everyone else, move to the 33 centimeter band, or run the standard as very nearly everybody does — and the guide was written using the standard values with a note that it is one field to change. That is a club decision, not an engineering one.

The most reproducible part of this project, though, is the shipping defaults. Three of them together produce a station that does absolutely nothing: transmit is disabled by default, digipeat mode is off by default, and — the one nobody finds — the factory transmit parameters do not match the factory receive parameters. A board with stock settings sits there hearing traffic and repeating none of it, with no error anywhere. Nothing was built in this project. The entire value was in reading primary sources — a repository's actual file tree, a regulation's actual text — instead of the articles about them, and then reporting a conclusion the requester did not want to hear.

The Thread Running Through All of It

Almost nothing in this work announced itself as broken. A firmware build that boots, responds, and radiates nothing. A node whose channel settings match and whose radio preset does not. A firmware that quietly clamps the value you set and reports success. An administrator key copied from a stale cache. A repeater

transmitting perfectly into a void because its clock was two years wrong. A LoRa APRS station whose factory defaults leave it inert.

That is why read-back verification, independent references, and passive monitoring earn their keep here more than anywhere else in this book. And it is where AI assistance provided the most leverage — not by knowing the answers, but by reading primary sources instead of secondary ones. Repository trees instead of blog posts about them. Firmware source instead of documentation. A device's actual reported configuration instead of an assumption about what it should be. The human corrections mattered too, and they belong in the record. The 433 MHz misidentification and the transmit-power advice were both wrong and both caught by an operator who knew the equipment. A book that only shows AI being right would be less useful, and considerably less believable, than one that shows the loop working.

Part 4C · Projects: Programming Assistance

Chapter 14 — Installing OpenClaw and Local Models

Chapter 3 explained what an AI agent is and why you might want one. This chapter tells the story of actually building one—twice. The first installation was done by Grok over many long sessions. The second, when I set up the Mac Mini as the model server and connected both agents to it, was done by Claude Code. Between the two, almost every common problem you'll encounter when setting up this kind of system showed up at least once.

Two Installations, Two Approaches

The OpenClaw platform described in Chapter 3 had to be installed on real hardware. I did it twice—first on a Dell PowerEdge server using Grok, then later on additional machines using Claude Code. The contrast between the two approaches is the story of this chapter.

The Dell R530 was a leftover enterprise server with 256 gigabytes of RAM. It seemed like the ideal platform until I learned the lesson covered in Chapter 3: without a GPU, model inference on it is painfully slow, so it works only as a host for the agent software while the model runs elsewhere. That architectural lesson cost me time, but the installation process taught me something more valuable.

What Grok Built

The initial installation of the production agent on the Dell server was done entirely through Grok. This was before I switched to Claude, and it was one of the more complex projects I attempted with Grok—setting up a Linux server from scratch, installing the container software that OpenClaw runs inside, migrating the test agent's memory and configuration to the new machine, and getting the local model server connected.

Grok walked me through every step. It told me how to configure the server's remote management interface so I could work on it from my desk without connecting a monitor. It guided me through clearing the old disk configuration and setting up new storage. It helped me install Ubuntu Linux, then Docker, then OpenClaw itself. When I needed to migrate the accumulated memory and configuration to the new machine, Grok explained how to copy everything over the network and reconnect it on the other end.

The process was not smooth. Enterprise server hardware has layers of configuration—security keys on the disk controller, network settings for the remote management interface, container networking that works differently from regular networking—and each one produced errors that Grok helped me work through. The local model server needed to be configured so that the OpenClaw container could reach it, which required understanding how container networking talks to the host machine. Grok knew the answers but couldn't execute the commands itself. I had to read each instruction, type it into the terminal, read the error message, paste it back to Grok, and wait for the next instruction. This paste-and-type cycle is what Chapter 4 calls the old workflow, and it works, but it's slow and error-prone.

What Claude Code Changed

When I later needed to reconfigure both agents to use the Mac Mini as their shared model server, I used Claude Code instead of Grok. The difference was immediate. Code connected to each machine over SSH, read the current configuration files, identified what needed to change, made the changes, tested the connections, and fixed the errors—all without me typing a single command.

Code edited the configuration files that tell each agent where to find its language model, pointing them at the Mac Mini's network address and port. When the first attempt failed because of a network permission issue, Code read the error, diagnosed it, adjusted the configuration, and tried again. When the model server wasn't responding on the expected port, Code checked whether the service was running, found that it wasn't, started it, and retested. The whole process—reconfiguring two agents to use a new model server—took less time than a single round of the paste-and-type cycle with Grok.

This is the chapter where the advantage described in Chapter 4—Claude Code as an operator rather than an advisor—becomes most visible. Server configuration is exactly the kind of work where small errors compound: a wrong port number, a mistyped network address, a permission that wasn't set. Each one produces an error message that requires diagnosis, correction, and retesting. Code handles the entire cycle internally, iterating until the system works. A human pasting error messages back and forth between a chat window and a terminal is doing the same work, just slower and with more chances to introduce new errors in the transcription.

What I'd Do Differently

If I were starting over, I would skip the Dell server entirely. A Raspberry Pi 5 is sufficient to run the agent software, and the Mac Mini handles the model serving. The right architecture is a cheap machine running the agent and a capable machine running the models, connected over the network—exactly the split laid out in Chapter 3.

But the real lesson from this project isn't about hardware. It's about the moment I realized that Code could operate directly on my other computers. That was a lightning bolt. It immediately started saving me hours of time, and it changed how I think about every project involving a Pi or any other networked machine.

Once Code has SSH access to a computer, it can do anything on that machine that you could do sitting at the keyboard. It can install software, edit configuration files, start and stop services, read error messages, change its approach, and keep going —all without you being there. When I later needed to install the Hermes agent platform on the Mac Mini, I just told Code to do it. It ran into errors, evaluated them, researched solutions, adjusted its approach, and completed the installation autonomously. I wasn't involved.

This applies far beyond agent platforms. Want to install an ADS-B aircraft tracker on a Pi? Ask Code. Want to set up Gpredict for satellite tracking? Ask Code. I've found articles describing Pi projects online, pasted the link into Code, and told it to read the article and make my Pi do what the article describes. It did it. No copying, no pasting, no researching each manual step, no wondering whether I typed a command correctly. Code reads the instructions and executes them, handling the errors along the way.

If you take one thing from this chapter: you no longer need to interact with your Pi or other computer directly unless you want to. Code can do it for you. This is the single most time-saving capability I've found in AI, and I wish I had realized it much sooner.

Chapter 15 — Automating the IC-9700

The Skunkworks satellite station uses an Icom IC-9700 as the primary radio for satellite contacts and a SatNOGS ground station for automated satellite telemetry reception. The SatNOGS system needs access to both the 2-meter and 70-centimeter bands simultaneously, but the IC-9700 only passes signals from both bands through to its rear-panel antenna connectors when the second VFO is active. If the second VFO is off, only the band selected on the primary VFO gets through. The SDR radio that SatNOGS uses for reception is connected to those rear-panel connectors, so without the second VFO running, SatNOGS can only hear one band at a time.

I needed a way to activate the second VFO automatically, remotely, and on a schedule—without anyone being at the radio.

The Constraint

The IC-9700 can be controlled by a computer through its CI-V interface—a serial protocol that Icom radios have used for decades. You send specific byte sequences through a cable, and the radio responds: change frequency, switch modes, activate VFOs. The radio also has a network connection for Doppler correction during satellite passes, and that connection was already in use. But the USB serial port on the back of the radio was available, and CI-V commands can be sent through it.

The problem was that the satellite station's PC shuts down when not in use. I needed something that would be on all the time to send the VFO commands on a schedule. The answer was an unused Raspberry Pi—always on, always connected, drawing almost no power.

What Code Did

I described the problem to Claude Code: I need the IC-9700's second VFO turned on automatically so both bands pass through to the SDR. Code wrote a Python program that sends CI-V commands through the USB cable to the radio. The sequence it came up with was clever: set VFO 1 to a 70-centimeter frequency, set VFO 2 to a 2-meter frequency, swap the two VFOs (which activates the second VFO), then swap them back so each band is on the correct VFO. The result is both VFOs active with the right frequencies, and both bands passing through to the SDR.

Code then deployed the program to the spare Pi. It SSH'd into the Pi, installed the program, connected to the radio through the USB serial port, tested the CI-V commands, and confirmed the radio responded correctly. Once it was working, Code set up a scheduled job that runs the program every hour to keep the configuration active—if someone manually changes the radio or it resets, the next hourly run puts it back the way SatNOGS needs it.

What I Didn't Need to Know

I don't know CI-V very well. I don't know the byte sequences that switch VFOs, set frequencies, or swap bands. I didn't need to. I described the behavior I wanted—both VFOs active, correct frequencies, running on a schedule—and Code handled the implementation. It knew the CI-V protocol, wrote the byte sequences, tested them against the actual radio, and fixed the ones that didn't work. The commands exist inside a Python program on a Pi that I've never edited.

This is the pattern from Chapter 1: sometimes AI teaches you what you need to know, and sometimes you don't need to know it at all. The CI-V commands are in the second category. The radio works. SatNOGS receives both bands. The hourly job keeps it running. That's all that matters.

Chapter 16 — Debugging the Satellite Rotator

The Skunkworks satellite station at the W2MMD clubhouse uses motorized rotators to point the antennas at satellites as they cross the sky. The rotators are controlled by Raspberry Pis running Node-RED, the visual programming environment described in Appendix B. We have two rotator controllers—one for the Yagi antennas and one for the tracking dish. The Yagi rotator worked. The dish rotator did not.

The Problem

The dish rotator's program had been adapted from the working Yagi program, but the two systems aren't identical. The dish is a 2.4 GHz antenna used for the ISS DATV project described in Chapter 16, and it needs to tilt to 180 degrees elevation to track overhead passes, whereas the Yagi system only elevates to 90 degrees. The dish also uses a different relay board whose inputs are reversed from the Yagi board—a command that turns a relay on for the Yagi turns it off for the dish, and vice versa. Somewhere in adapting the program for these differences, things had gone wrong. Buttons that should have moved the antenna did nothing. The satellite tracking system wasn't following passes. The elevation display was reading half of what it should have been. I spent hours clicking through the Node-RED editor, checking nodes one by one, and couldn't find the problem.

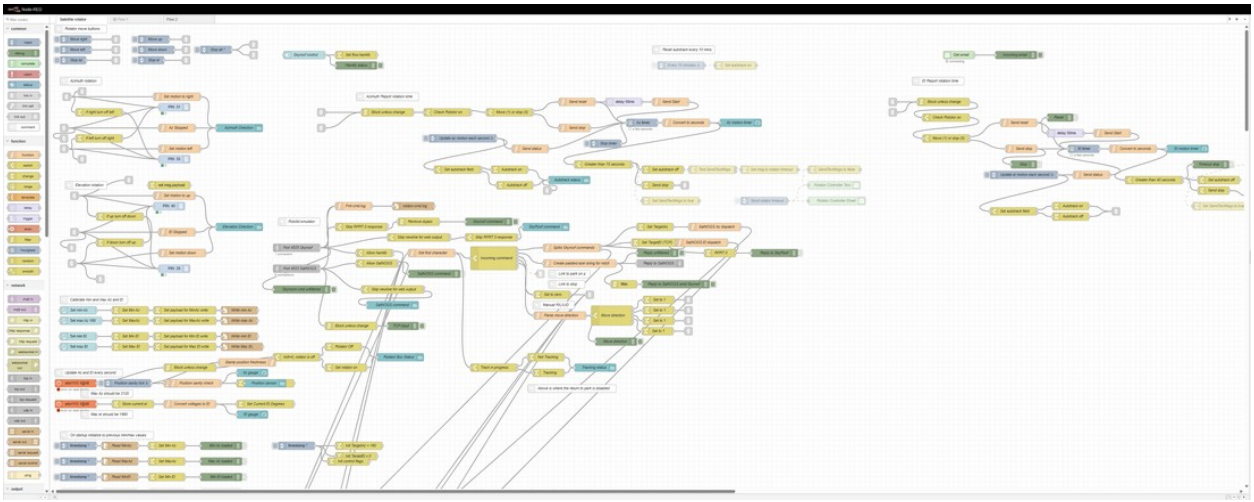


Figure 6 - The Rotator Controller Node-Red Code

What Code Did

I gave Claude Code access to both Pis—the working Yagi controller and the broken dish controller. Code connected from my home to the clubhouse network through a VPN, pulled the programs from both machines, and compared them.

As described in Appendix B, Node-RED looks graphical to the user but is stored as text files underneath. Code read both programs as text and found 47 differences.

The biggest category was the reversed relay board: dozens of on and off commands throughout the program were backwards because the dish's relay board responds to the opposite signal from the Yagi's board. Commands that should have said "move" were saying "stop," and vice versa, in more than twenty places. Other bugs included the elevation range still set to the Yagi's 90-degree limit instead of the dish's 180 degrees, and math errors in the formulas that convert sensor readings to degrees. Each of these bugs was invisible in the graphical editor because the nodes looked identical whether they contained the right value or the wrong one.

Code applied all 47 fixes in one pass, deployed the corrected program to the dish controller, and the rotator worked.

A Later Fix

A few weeks later, the rotator developed a different problem: the position sensors were producing wild readings at the extremes of their travel. Code connected to the clubhouse again, read the current program, and added limits to the conversion formulas so that any reading beyond the antenna's physical range gets pinned to the boundary instead of displaying an impossible number. The fix took a few minutes and was deployed remotely without anyone visiting the clubhouse.

Why This Matters

This is one of the shorter projects in the book, but it demonstrates something important. I had spent hours looking at the visual program and couldn't find the bugs. Code read the same program as text and found all 47 of them in minutes. It could compare the working version against the broken version, identify every difference, determine which differences were intentional and which were errors, and fix them all at once. It did this from my home, through a VPN, on a Pi at the clubhouse twenty miles away.

A Node-RED node that sends a "move" command looks identical to one that sends a "stop" command in the graphical editor. The only way to tell them apart is to click on each one and read its properties. Code doesn't need to click. It reads the whole program at once.

Chapter 17 — The Pi Fleet Dashboard: A Project That Grew by Asking

Most of the projects in this book began with a clear goal. This one began with an annoyance, and then kept growing every time I thought of something else it could do. It is the clearest example I have of a pattern worth naming: when the tool costs almost nothing to extend, you stop rationing your ideas.

The Annoyance

I had accumulated a fleet of Raspberry Pis — some at home, some at the clubhouse, on two different subnets. Bob running the agent sandbox. Arnold running production. A SatNOGS station. WSPR beacons. Rotator controllers. Mesh nodes. Several general-purpose Pis that did whatever I had last set them up to do.

The problem was that I could no longer remember which was which. Finding out what a given Pi was running meant opening a terminal, connecting over SSH, and poking around — checking which services were enabled, what was listening on which port, what was in the crontab. Thirty seconds of work, but thirty seconds every single time, and I was doing it constantly. What I wanted was to type a Pi's address into a browser and have it simply tell me what it was.

How It Grew

The first request was exactly that: a small web page, served by each Pi, showing what it is and what it is doing. Claude sketched the design in a few minutes. And then, because the marginal cost of asking for one more thing was effectively zero, I kept asking.

My first addition was cosmetic and turned out to be the most useful one. I asked for the color scheme to be configurable per Pi, so that each dashboard would look different — easier to keep track of when you have six browser tabs open. Claude took that further than I had: a configurable emoji icon in the header as well, and the favicon colored to match, so the tabs themselves are distinguishable before you read a single word. A satellite dish for SatNOGS, a robot for Arnold, an antenna for the WSPR Pi.

Then I asked the question that opened the floodgates: what else could we do to make this useful? The answer was a long list, and I took most of it. An inline-editable friendly name and description, saved back to a configuration file on the Pi itself, so I could rename a machine from the browser instead of editing a file over SSH. A list of the systemd services and cron jobs that start automatically, each with a restart button. System metrics — processor load, memory, disk, temperature, network throughput, uptime — refreshing every thirty seconds without reloading the page. An undervoltage warning, which is the single most common silent failure on a Pi and produces symptoms that look like everything

except a power problem. A free-text notes field for the things you would otherwise forget, like which USB port the SDR is plugged into.

For the Pis that run things manually rather than automatically, a grid of task cards with start, stop, and restart buttons. That one raised a design question I had not anticipated: if I start something like a receiver or a ground-station program from the dashboard, how does the dashboard know where its web interface is? Claude proposed a hybrid detection scheme — watch for newly opened ports and guess. I did not like it. Since every task already needed its startup command listed in the configuration file, the port could simply be listed there too, and the dashboard could poll that one port and reveal an “Open UI” link the moment it went live. Simpler, deterministic, and no guessing. That exchange is worth recording because the human made the design better, not the AI.

The last feature was the most ambitious: each dashboard discovers the other Pis on the network and links to them. The design question there was whether to scan on demand, cache results, or maintain a central registry. I chose on-demand and asynchronous — the page loads immediately with everything else, and the peer list fills in underneath as machines are found. Any Pi becomes an entry point to the whole fleet.

Finding the Pis to Deploy To

Building the dashboard was only half the problem. It had to get onto a dozen machines whose addresses I did not reliably know, because they are assigned by the router and change. I asked whether it was possible to scan for the Pis by their hardware addresses instead, which do not change. It was. The deployment script sweeps both subnets, reads the address table, matches each machine against a list of known hardware addresses, and pushes the files to whatever address that Pi currently has.

Along the way I asked what Ansible was, having seen it mentioned as the standard tool for exactly this kind of job. Claude explained it and then told me not to use it — that for a handful of machines on a private network, a short script doing the same copy-and-restart loop would accomplish the same thing with nothing new to learn. That is worth noting: I asked about a tool and was talked out of it, with reasons. AI that only ever says yes is less useful than AI that tells you when something is overkill.

Design in Chat, Build in Code

The entire design happened in a chat conversation — no code written, no files created, just a back-and-forth about what the thing should do. At the end I asked Claude to turn the whole discussion into a single complete specification that could be handed directly to Claude Code: the dashboard server, the web page, the service definition, a sample configuration file, and the deployment script, with the behavior of each spelled out. Code then built it and pushed it to the Pis.

That division turned out to be the right one, and it recurs throughout this book. Chat is where you think — where an idea can be proposed, questioned, and revised cheaply, and where changing your mind costs nothing. Code is where you build. Moving between them deliberately, rather than trying to design and implement in the same breath, produces better results than either one alone.

What the Project Actually Demonstrates

Not one of the features beyond the first was in my head when I started. The colored headers, the emoji icons, the undervoltage warning, the editable notes, the task cards, the port polling, the peer discovery, the hardware-address deployment — every one of them arrived because the cost of asking was low enough that I asked. With conventional development, each addition would have meant an evening of work, and I would have stopped after the second one. Here the sequence of revisions was itself the design process.

That is the practical lesson. When building something becomes cheap, your imagination becomes the constraint rather than your time. The right response is not to plan more carefully. It is to keep asking for the next thing and see how far it goes.

Part 4D · Projects: Large-Scale Analysis

Chapter 18 — SDR Meets AI: Signal Analysis Without a DSP Degree

A software defined radio turns radio signals into numbers. Once a signal is numbers in a file, it is data — and analyzing data is something AI does well. That single observation opens up a category of work that used to require either specialized software or a background in digital signal processing.

The pattern is always the same. You capture what came out of the antenna, hand the file to Claude Code, and describe what you want to know. Code writes the analysis program, runs it, reads the output, adjusts its approach when the first attempt does not work, and explains what the numbers mean. You do not write the code and you do not need to know the mathematics behind it.

Identifying an Unknown Signal

You hear something on the waterfall that you cannot identify. Capture a short recording of it and ask what it is. Code can measure the bandwidth, look at how the energy is distributed, check whether the signal is continuous or bursty, measure the symbol rate if it is digital, and compare all of that against the

characteristics of known modes. It will not always give a definitive answer, but it will narrow the field considerably and tell you what evidence points where.

Measuring Your Own Station

Point the SDR at your own transmitter and you can measure things that are otherwise hard to see: how clean the signal is, whether there is spurious output, how the power distributes across the occupied bandwidth, whether an amplifier is producing distortion products. These are measurements a service monitor would give you, derived instead from a captured file and a program written on request.

Validating a Weak Signal Decode

When a decode looks marginal, the question is whether the software actually found a signal or whether it produced a false decode from noise. Code can examine the captured audio or IQ, measure the signal-to-noise ratio in the relevant bandwidth, and tell you whether there was enough signal present to justify believing the result. This is the same reasoning that closed out the ISS DATV project in the next chapter, applied at a smaller scale.

Antenna Patterns and Doppler

Drive around with a receiver and a GPS, log signal strength against bearing, and you have the raw material for an antenna pattern. Code can turn that log into a polar plot and compute the front-to-back ratio and beamwidth. Similarly, a capture of a satellite pass contains the Doppler curve as a measurable property of the signal — Code can extract it, fit it against the predicted curve from the orbital elements, and tell you whether your tracking software's correction is actually working.

Interference Hunting

The most practical application at the clubhouse is finding noise sources. Capture the spectrum in different directions, at different times of day, with different equipment switched on and off, and let Code compare the captures to identify what changed and where it came from. That approach grew into the automated RF survey system described in Chapter 19, but it started as a handful of manual captures and a request to compare them.

What This Actually Requires

An RTL-SDR dongle, an antenna, and software that can save a raw capture. The dongle costs about thirty dollars. Everything else is the conversation. You do not need to install specialized analysis software, learn its interface, or understand the algorithms it implements — you describe what you want to know and Code writes what it needs to find out.

Chapter 19 — Decoding the ISS: When the Answer Is No

This chapter was supposed to end with decoded video from the International Space Station. It doesn't. What it ends with is something more useful: a methodical demonstration of how Claude Code can take a raw IQ recording, run it through every plausible decode path, diagnose exactly why decoding failed, and tell you what to fix. The answer turned out to be the antenna, not the software. Getting to that answer took one evening and a conversation with an AI that never got tired of trying the next parameter combination.

The Capture

On May 27, 2026, the ISS made a near-zenith pass over the W2MMD clubhouse—86.5 degrees peak elevation, 418 kilometers minimum slant range. This was about as good a pass geometry as you can get from southern New Jersey. The SDR was tuned to 2395.044 MHz with a 3 MHz capture bandwidth, and Skyroof handled Doppler tracking on the SDR's local oscillator during the pass. The result was a 6.2 gigabyte RF64 WAV file containing approximately nine minutes of IQ data.

I handed the file to Claude Code, running from a Code session over SSH on the workspace Raspberry Pi. What followed was a systematic attempt to extract video from that recording using every DVB decoding tool available.

Confirming the Signal Was Real

I already knew the recording contained a genuine ISS signal. During the pass I had watched the Doppler shift on SDR Console—the carrier sliding steadily down in frequency as the station approached and receded, exactly the signature a real spacecraft signal produces and nothing terrestrial does. That was my confirmation in real time.

What Code did was reproduce that verification independently, as a cross-check, and it was worth having. A Python script measured mean power across the capture in time windows and correlated those measurements against ISS elevation and reciprocal slant distance computed from the spacecraft's TLE. Power tracked elevation at a correlation of +0.50 and tracked inverse distance at +0.52, with the peak power coinciding with the time of closest approach. A second script verified the Doppler-correction model against the recording's strongest in-band tones. Both agreed with what I had seen live: the signal was genuinely ISS-borne. This mattered because everything that followed was a negative result, and having two independent confirmations—my eyes on the waterfall and Code's correlation analysis—meant nobody could argue the recording was just noise. It wasn't noise. It was ISS HamTV—just not enough of it.

Three Decoders, Zero Locks

Code built and ran three independent DVB decoders against the recording. Each one was swept across every plausible combination of parameters. None of them locked.

The first decoder was LeanDVB, built from the pabr/leansdr project, targeting DVB-S. Code drove it through a shell sweep script that tried every plausible HamTV symbol rate—125k, 250k, 333k, 500k, 1M, 1.5M, and 2 Msymbols per second—with three different lock-aggressiveness settings and tune offsets of plus and minus 5, 20, and 50 kHz around the nominal center frequency. Every combination produced zero lock events. The best signal quality metric reported anywhere in the sweep was 5.0 dB, below the 5 to 7 dB threshold needed for DVB-S QPSK half-rate lock.

The second decoder was a development branch of LeanDVB with DVB-S2 and LDPC support enabled. Code swept all 32 DVB-S2 modulation and coding configurations across multiple symbol rates and both frame sizes. The decoder produced rising frame-lock counters, which looked promising for about thirty seconds until Code noticed that the lock count scaled linearly with symbol rate—the classic signature of random noise occasionally producing the 26-bit start-of-frame sync pattern by chance. No valid physical-layer signaling was ever decoded.

The third decoder was gr-dvbs2rx, Igor Auad's GNU Radio DVB-S2 receiver. Code installed GNU Radio 3.10.12 from the system packages, built gr-dvbs2rx from source, and tested it against a 60-second slice from the middle of the pass across four symbol rates and five QPSK configurations. Every run reported lock false, SNR null—the SNR estimator never had enough signal structure to produce a value. Code even drove the Qt GUI headlessly under Xvfb and captured screenshots of the waterfall display. The waterfall showed broad in-band energy that tracked the pass elevation, but no clean rectangular DVB-S spectral shape and no constellation cluster formation.

Every transport-stream output file across all three decoders—every .ts file in the decode attempts directories—was zero bytes. Not one MPEG-TS packet was produced by any tool, any parameter combination, any symbol rate.

The Constellation That Told the Story

The diagnostic that closed the question was a short Python script that Code wrote to extract complex samples at exactly 2.0 and 1.3 megasamples per second—the two published HamTV symbol rates—applied root-raised-cosine matched filtering, and plotted the result as a scatter diagram on the IQ plane.

The output was a pure circular Gaussian blob centered on the origin. At both symbol rates. A locked DVB-S signal, even one running close to threshold, would show four distinct clusters at the corners of a square—the four QPSK symbol positions. This recording showed nothing of the kind. There was no symbol

structure for any demodulator to lock onto. The blob ruled out wrong parameters, wrong symbol rate, and wrong DVB version as causes of failure. The signal was simply too weak.

What It Meant

The session's conclusion was precise. The recording contained genuine ISS HamTV—the elevation correlation proved it. The realistic signal-to-noise ratio delivered to the demodulator was 2 to 6 dB, right at or below the DVB-S lock threshold of 5 to 7 dB. For comparison, the confirmed amateur HamTV decode by K4KDR in December 2025 used a one-meter S-band dish providing approximately 22 dBi of gain. The W2MMD setup was 6 to 10 dB short of that. Closing the gap is an antenna and front-end problem, not a decode-software problem.

Code reached this conclusion in one session. It built three decoders, ran hundreds of parameter combinations, wrote custom diagnostic scripts, captured GUI screenshots headlessly, and produced a definitive answer: the toolchain works, the parameters are correct, the signal is real, and the antenna is too small. A human working alone, building each decoder manually, writing sweep scripts, interpreting ambiguous lock indicators, would have spent days or weeks reaching the same conclusion—or might have given up convinced it was a software problem.

Why a Failure Is Worth a Chapter

This project did not produce decoded video. It produced something more valuable for this book: a clear demonstration of what AI-assisted signal analysis looks like when applied to a real-world problem with a real-world negative result.

Code did not guess. It did not try one decoder and give up. It built every available tool, swept every plausible parameter space, wrote custom diagnostics when the standard tools' output was ambiguous, and arrived at a conclusion that was not only correct but actionable. The W2MMD Skunkworks group now knows exactly what it needs: a larger dish, or a lower-noise front end, or both. The software is ready. The infrastructure is ready. The next near-zenith pass with a bigger antenna will tell a different story.

That is the value of AI in signal analysis. Not that it always succeeds, but that when it fails, it fails informatively. It tells you why, and it tells you what to change.

Chapter 20 — Projects in Progress

The projects in earlier chapters are complete or at a stable stopping point. These two are still under active development. I'm including them because they illustrate something the finished projects don't: what it looks like when AI-assisted engineering is still in the middle of the problem, with open questions and unsolved issues alongside the parts that work.

The EmComm Situation Recorder

This is a Raspberry Pi appliance designed to sit on an emergency communications net and keep a timestamped, verbatim-backed log of everything said and reported during an incident. It listens to voice traffic from handheld radios on multiple audio channels, decodes APRS packet positions from a separate receiver, takes GPS from its own module, and answers plain-English questions on a touchscreen — “where is WB2MNF,” “what was AD2CS's last report,” “who is overdue.” Every event lands in a tamper-evident log; nothing is ever edited or deleted, and corrections are recorded as new events that supersede old ones. The original audio for each transmission is retained alongside its transcript, so a disputed entry can always be played back rather than argued about.

The AI in the system is deliberately narrow. Speech recognition turns audio into text. A voice activity detector separates speech from squelch noise. A text-to-speech engine reads answers back to the operator. Everything above that is deterministic — callsign matching, event extraction, and the question engine are all rule-based lookups against the database, not AI-generated responses. That constraint was chosen deliberately: a language model in the evidence path can produce a fluent, confident, wrong answer, and an incident log is evidence. The system also matches garbled callsigns only against operators actually registered for that event, never a wider database, because a bad transcription can plausibly match the callsign of someone who was never there. Everything runs completely offline — models, maps, and reverse geocoding so positions read “on Winterberry Way” rather than as decimal coordinates.

The genuinely surprising lesson was that the AI was the easy half. Almost every real failure lived in the analog and RF layer, and almost all of them were silent — the software kept running and produced confident, wrong output. Audio driven a few decibels too hot didn't throw an error; it fused two operators' callsigns into one. A four-millisecond click when an operator released their push-to-talk button swung the measured audio level by 14 dB and made a perfectly set radio report as too loud the instant they let go. Identical USB audio dongles sitting a centimeter apart meant a swapped cable sent packet data to the voice transcriber, which duly invented words out of modem tones — now detected by testing whether the audio has a flat envelope, since data measures zero variation while speech measures about 8.6 dB.

Most instructive of all: the “correct” audio level had been calibrated from studio engineering convention, and measuring against real radio traffic showed it was simply wrong. Callsigns survived transcription 89 percent of the time at one level but only 50 percent at the level the system had been advising operators to use. The build now includes a setup page that measures each channel and tells the operator which way to turn the volume knob — because a level problem that produces plausible-looking transcripts is far more dangerous than one that produces an obvious error. The useful story here isn't that AI solved emergency communications logging. It's that a small, deliberately unambitious amount of AI, wrapped in careful measurement and an evidentiary discipline borrowed from incident command, produced something an operator could actually trust. One problem remains genuinely open: the APRS receiver decodes its own local beacon flawlessly but has never decoded a distant station, which the evidence points at the RF path rather than the software.

The RF Noise Survey System

The clubhouse sits on a site shared with the Gloucester County 4-H fairgrounds, surrounded by potential noise sources: switching power supplies inside the building, pumps and lighting from the fairgrounds, overhead power lines, nearby cell towers, and the club's own transmitters — WSPR beacons, Meshtastic nodes, a Winlink station, the SatNOGS receiver, and the Discovery Dish. All of these contribute to the noise floor that limits what the station can hear. The question is which sources can be eliminated, and the first step is figuring out where the noise is coming from.

Claude Code designed and built an automated RF survey system that uses an SDR receiver paired with a rotatable directional antenna. The system sweeps 360 degrees in steps, capturing a slice of spectrum at each heading, computing the noise power, and logging the result with the heading and frequency in a database. After a full sweep, the data can be pulled over the VPN to the home network for AI analysis — looking for directional noise sources, interference patterns, and changes between sweeps taken at different times of day.

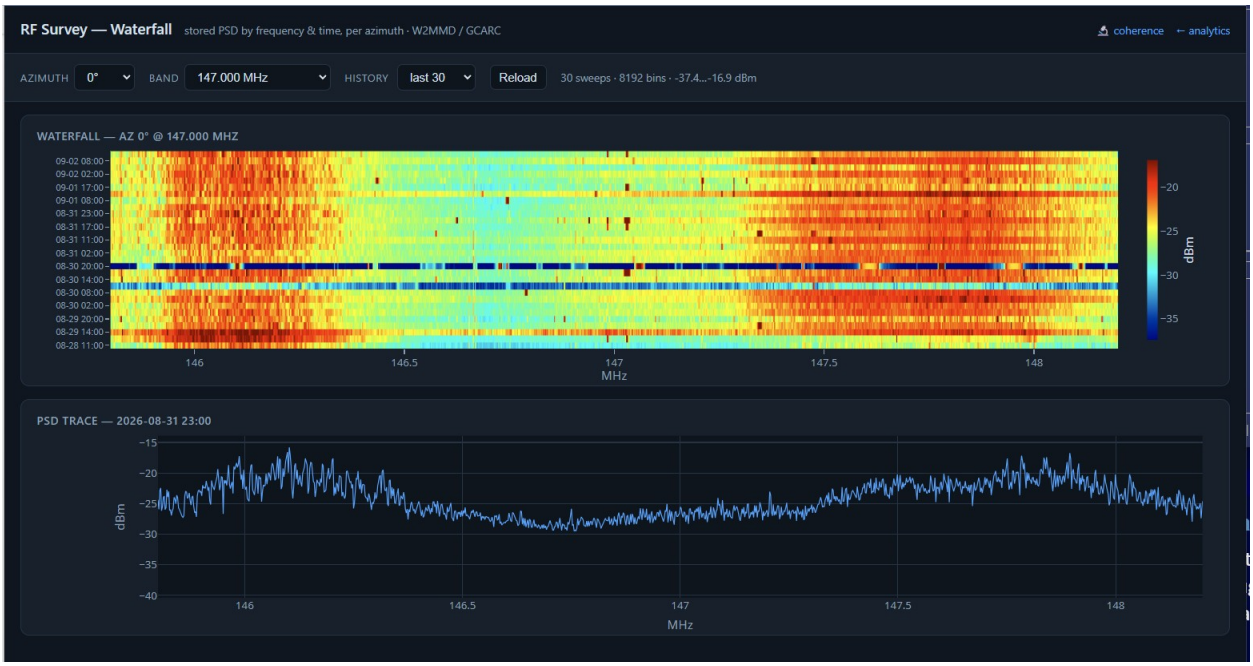


Figure 7 - The RF Survey Waterfall

The survey application coordinates with the SatNOGS ground station that shares the same rotator. It checks SatNOGS's schedule and pauses during satellite passes, then resumes the sweep when the pass is complete. It can run a single-band sweep in a few minutes or a full wideband scan — stepping through the entire VHF/UHF range in 2.4 MHz slices at each heading — as an overnight unattended run. Code also designed a portable version of the system for field surveys away from the clubhouse, using a small antenna rotator, a GPS module for automatic location tagging, and a WiFi hotspot so the same software works without a network connection. The portable kit is intended for identifying noise sources at members' home stations and for community events like Field Day where the operating environment isn't known in advance.

Both the recorder and the survey system are works in progress. Including them here is partly a status report and partly a preview of what's coming — but mostly an honest acknowledgment that not every AI project wraps up in a neat chapter. Some are still in the middle.

Part 5 · More Tools

Chapter 21 — NotebookLM: AI That Only Knows What You Tell It

Most of the AI in this book generates responses from a vast body of training data, which means it occasionally produces confident, plausible answers that are simply wrong. For most tasks the tradeoff is acceptable: you verify the important claims and move on. But there is a category of work where hallucination is not acceptable at all, where you need the AI to answer strictly from specific documents and nothing else. That is what Google's NotebookLM does, and it has become one of the most useful tools in my workflow. I've put it near the end of the book because it's a specialized complement to everything that came before, not a starting point—but for the right job, nothing else comes close.

What Makes It Different

NotebookLM is an AI research tool built around a simple constraint: it only knows what you upload to it. You create a notebook, load it with sources—PDFs, web pages, YouTube videos, audio recordings, Google Docs—and the AI answers exclusively from that material. It will not guess. It will not fill gaps with general knowledge. If the answer is not in your sources, it tells you it doesn't know. This single property eliminates the hallucination problem for any question that can be answered from your documents.

The sources can be substantial. You can load dozens of documents into a single notebook, and NotebookLM will index and cross-reference them. It cites its sources in every response, pointing you to the specific page, paragraph, or timestamp where it found the answer. You can verify any claim by clicking the citation and reading the original.

Ham Equipment Manuals

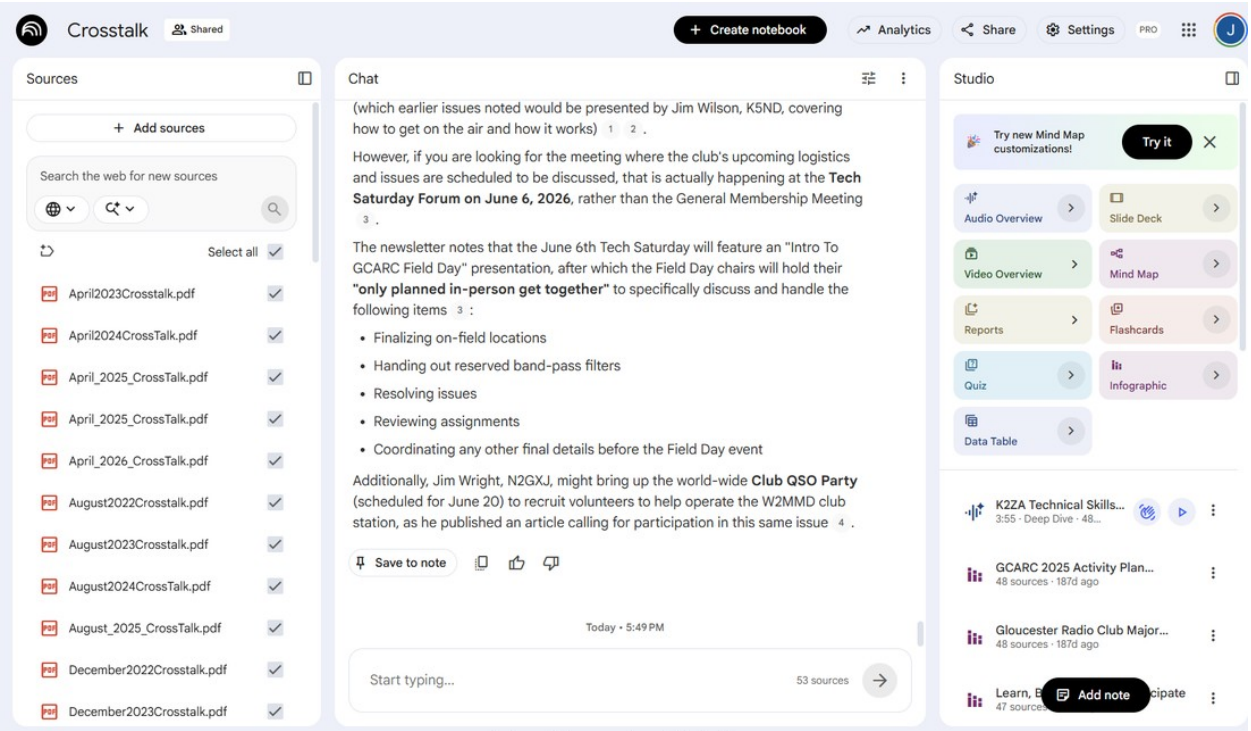
The most immediately practical use I've found is building a notebook of ham equipment manuals. I loaded the operating manuals for several of my radios and accessories into one notebook. Now instead of searching through a 200-page PDF trying to find the menu setting for a specific feature, I ask NotebookLM: "How do I set the IC-9700 to split-tone operation?" or "What is the default CI-V address for the IC-9700?" It answers from the manual and shows me exactly where to find it.

The real power emerges when you load manuals for multiple devices into the same notebook. I loaded both the Yaesu FT-991A manual and the WSJT-X documentation, then asked how to configure WSJT-X for the 991A—the correct COM port settings, audio routing, CAT control parameters. NotebookLM pulled the

relevant details from both documents and assembled a coherent answer that would have taken me thirty minutes of cross-referencing to piece together manually. Every claim was cited back to one manual or the other. No hallucination, no guessing, no invented settings.

Club History

I loaded four years of CrossTalk, the GCARC newsletter, into a notebook. Years of club history—meeting minutes, activity reports, member contributions, event announcements—became instantly searchable in a way that no PDF search or file browser could match. I can ask "When did the club first discuss satellite operations?" or "What activities did we run in 2024?" and get specific answers with citations to the exact issue and page. For a club president trying to understand the history of a program or track how an initiative evolved, this is invaluable.



Meeting Minutes from Recordings

NotebookLM accepts audio recordings as sources. I've uploaded recordings of club meetings, and it produces transcripts and meeting minutes directly from the audio. The quality is good enough for a first draft that needs only light editing. For anyone who has ever been the reluctant secretary at a club meeting, this alone is worth knowing about.

Consolidating Video Sources

YouTube videos can be loaded as sources alongside documents. If you're researching a topic—say, building a mesh network, or setting up SatNOGS, or

understanding EME propagation—you can collect the best YouTube tutorials, load them into a notebook alongside the relevant documentation, and ask NotebookLM to consolidate the information. It will pull from both the video transcripts and the written documents to give you a unified answer, cited to the specific video timestamp or document page.

Producing Deliverables

NotebookLM can produce structured output from your sources: summaries, study guides, FAQs, timelines, briefing documents, and—notably—audio podcasts generated from the material, where two AI voices discuss the content in a conversational format. It can also generate slide deck outlines and research papers. These are produced entirely from your uploaded sources, not from general knowledge, which means the output is grounded in material you've vetted.

When to Use NotebookLM vs. Claude

The decision is straightforward. Use NotebookLM when the answer must come from specific documents and nothing else: equipment manuals, club archives, meeting recordings, technical standards, regulatory documents. Use Claude when you need general knowledge, code generation, creative writing, system administration, or any task where breadth of knowledge matters more than strict source fidelity. They are complementary tools. NotebookLM is the best product I've found when working with a fixed, limited set of source material. Claude is the best product I've found for everything else.

Appendices

Appendix A — The Raspberry Pi: A Ham's Swiss Army Knife

If you walk into the W2MMD clubhouse and start counting Raspberry Pis, you'll lose track before you're done. There are dozens of them, tucked into corners, zip-tied to shelves, plugged into software-defined radios, sitting in weatherproof enclosures near the tower. Each one does exactly one thing, and that's the point.

The Raspberry Pi has become the single most useful piece of hardware in amateur radio — not because it's powerful, but because it's cheap enough and small enough to dedicate to a single purpose. When you need to do something new, you don't repurpose an existing computer. You buy another Pi. At thirty-five to eighty-five dollars depending on the model, the economics of single-purpose computing change everything about how you build a station.

This appendix covers what you need to know about the Raspberry Pi to follow the projects in this book. It is not a comprehensive Linux tutorial. It's the minimum necessary to get a Pi on your network, talk to it, and run the programs that Claude Code will write for you.

Why the Pi Matters

A traditional ham shack computer is a general-purpose machine running a general-purpose operating system. It does logging, digital modes, rotator control, SDR reception, email, web browsing, and whatever else you need, all on one box. This works until it doesn't. The logging program conflicts with the SDR software. A Windows update breaks the rotator controller. You need to reboot for one application, but another application needs to stay running.

The Pi approach is different. Each task gets its own computer. The SatNOGS satellite ground station runs on its own Pi, connected to its own SDR and its own antenna. The WSPR transmitter runs on a different Pi. The APRS weather station runs on a third. The ADS-B aircraft tracker runs on a fourth. They don't interfere with each other because they don't share anything. If one crashes, the others keep running. If you need to rebuild one, the others are unaffected.

At the clubhouse, this philosophy has produced a fleet of Pis that collectively handle satellite telemetry collection, satellite image reception, weather data ingestion into the APRS network, WSPR beacon transmission, ADS-B aircraft tracking, SatNOGS ground station operations, GOES weather satellite image processing, and Node-RED automation for rotator control and data processing. No single machine does more than one or two of these tasks. The total hardware cost for all of them is less than a single high-end PC.

The Pi Family

There are several models of Raspberry Pi in current use. Each has a different price point and capability level, and choosing the right one for a given task is mostly about matching the workload to the hardware.

Raspberry Pi Zero

The Pi Zero is the smallest and cheapest member of the family. It's about the size of a stick of gum and costs around fifteen dollars. It has WiFi but no Ethernet port, which means it can connect to your network wirelessly but can't be plugged into a wired LAN without an external USB adapter. It also requires special adapters for a monitor, keyboard, and mouse, which makes initial setup slightly more involved than the larger models.

The Pi Zero is best suited for truly simple, low-power, embedded applications where size and cost matter more than performance. A simple telemetry beacon, for example, doesn't need much computing power and benefits from the Zero's tiny footprint. But for anything involving serious data processing, SDR decoding, or AI workloads, you'll want a larger model.

Raspberry Pi 3

The Pi 3 is the workhorse. It has Ethernet, WiFi, four USB ports, HDMI output, and enough processing power for the majority of ham radio applications. It's the model we used for the WSPR transmitter build sessions — eighteen of them, assembled over two Tech Saturdays, each one running a TAPR WSPR board and transmitting on multiple HF bands. A Pi 3 handles the WSPR encoding, timing, and frequency synthesis without breaking a sweat.

At roughly thirty-five dollars, the Pi 3 is cheap enough that you don't think twice about dedicating one to a task. Need an APRS iGate? Pi 3. Need a Node-RED automation controller? Pi 3. Need a web server to display satellite pass predictions? Pi 3. The default answer to "what should I run this on?" is almost always a Pi 3 unless there's a specific reason to go bigger.

Raspberry Pi 4

The Pi 4 is a step up in both price and capability. It comes with more RAM options (up to 8 GB), a faster processor, USB 3.0 ports, and gigabit Ethernet. The additional horsepower matters for applications that involve real-time signal processing: running an SDR receiver with significant bandwidth, decoding digital satellite transmissions, or serving a SatNOGS ground station that needs to handle multiple passes in rapid succession.

The Pi 4 can also be extended with an M.2 solid-state drive through a HAT (Hardware Attached on Top) adapter board. This is important for applications that write large amounts of data continuously. At the clubhouse, we use a Pi 4 paired with an M.2 drive for the GOES weather satellite receiver and image processor. The GOES system receives a continuous data stream from the SDR, processes the

raw signal into images, and stores them to the M.2 drive. An SD card would not survive this workload for long, but the M.2 drive handles it without issue. That Pi lives in the gray Argon case on the Pi table in the clubhouse — the case provides both M.2 mounting and passive cooling.

The Pi 4 also draws more power and generates more heat than the Pi 3, so it benefits from a case with a heatsink or a small fan, especially under sustained workloads like continuous SDR reception.

Raspberry Pi 5

The Pi 5 is, for practical purposes, as powerful as a desktop PC. With a significantly faster quad-core Cortex-A76 processor, PCIe 2.0 support, and up to 16 GB of RAM, it can handle workloads that would have required a full-size computer a few years ago. This is the platform I use for the OpenClaw AI agent Bob, which runs on a Pi 5. It's also the platform of choice for SDR Connect and other programs that dump large volumes of data and need substantial processing power.

The Pi 5 includes native support for NVMe storage through its PCIe interface. NVMe stands for Non-Volatile Memory Express — it's a type of solid-state drive that connects through a high-speed PCIe bus rather than through USB. NVMe drives are dramatically faster than USB-connected SSDs and orders of magnitude faster than SD cards, both in read/write speed and in the number of simultaneous operations they can handle. For a Pi 5 running an AI agent, an SDR receiver, or any I/O-intensive application, an NVMe drive in an M.2 slot is a significant upgrade over an SD card. Several Pi 5 cases include M.2 NVMe mounting along with active cooling — a fan or large heatsink — which the Pi 5 needs under sustained load.

Pricing for the Pi 5 has increased significantly in 2026 due to global LPDDR4 memory shortages driven by AI infrastructure demand. As of April 2026, the current prices are: 1 GB model at $45, 2 GB at $65, 4 GB at $85, 8 GB at $125, and 16 GB at $305. The 1 GB and 2 GB models work well for headless network appliances and lightweight tasks. For AI agents and SDR work, the 4 GB or 8 GB models are the practical choices. The Pi Zero and Pi 3, which use older LPDDR2 memory, have not been affected by these price increases.

Model

RAM

Connectivity

Price

Storage

Typical Ham Use

Pi Zero

512 MB

WiFi only

~$15

SD only

Simple beacon, sensor node, low-power embedded

Pi 3

1 GB

WiFi + Ethernet

~$35

SD only

WSPR transmitter, APRS iGate, Node-RED, web server

Pi 4

1–8 GB

WiFi + GigE + USB3

$35–$95

SD, M.2 via HAT

GOES receiver, SatNOGS, SDR with high bandwidth

Pi 5

1–16 GB

WiFi + GigE + USB3 + PCIe

$45–$305

SD, NVMe via PCIe

OpenClaw agent, SDR Connect, AI workloads, image processing

Preparing an SD Card

Before a Pi can do anything, its SD card needs to be loaded with an operating system. The tool for this is the Raspberry Pi Imager, a free program from the Raspberry Pi Foundation that runs on Windows, Mac, or Linux. You download it from raspberrypi.com/software, install it, insert a micro SD card into your computer, and follow the prompts.

The Imager does more than just write an operating system image to the card. Before it starts writing, it offers a settings screen where you can preconfigure several things that would otherwise require connecting a monitor and keyboard to the Pi for first-time setup. This is critical because most ham radio Pis will run headless — no monitor, no keyboard, no mouse — from the moment they're powered on.

In the Imager settings, you can set the Pi's hostname — the name it will use on the network, like "bob" or "wspr-tx-1". You can create a username and password for

SSH login. And you can enter your home WiFi network's SSID and password so the Pi connects to your network automatically on first boot. With these three settings configured, you can insert the card into the Pi, plug in power, wait about a minute, and SSH in from your laptop. No monitor needed, ever.

A detailed walkthrough of the Raspberry Pi Imager is beyond the scope of this appendix, but the Raspberry Pi Foundation's own documentation at raspberrypi.com/documentation covers the process step by step with screenshots. There are also excellent video tutorials on YouTube — search for "Raspberry Pi Imager headless setup" and you'll find several that walk through every screen. The process takes about five minutes.

IP Addresses: How Your Pi Gets Found on the Network

Every device on your home network — your laptop, your phone, your Pis — has an IP address. This is a four-number sequence like 192.168.1.50 that uniquely identifies the device on the network. When you SSH into a Pi, you're connecting to its IP address.

IP addresses are assigned by a DHCP server, which in most home networks is your router. When a Pi boots up and connects to your WiFi or plugs into Ethernet, it sends a request to the router saying "I need an address." The router picks an available address from its pool and assigns it. This happens automatically and invisibly.

The problem with automatic assignment is that the address can change. If you reboot your router, or if the Pi's DHCP lease expires, it might get a different address the next time. This breaks your saved SSH sessions, your scripts, and any services that reference that address.

The fix is a DHCP reservation, sometimes called a static lease. You tell your router to always assign the same IP address to a specific device, based on the device's MAC address — a unique hardware identifier burned into every network interface. This is typically configured in your router's administration page. Once set, Bob is always at the same IP address regardless of power cycles or router reboots.

If you're not sure how to set up DHCP reservations on your specific router, ask Claude. Give it the router's make and model and it will walk you through the steps. This is a one-time setup that pays off permanently.

Linux: The Operating System

Every Raspberry Pi runs Linux — specifically, a distribution called Raspberry Pi OS, which is based on Debian. If you've never used Linux before, this can feel intimidating. It shouldn't be. You need a small number of commands to be productive, and you'll learn them quickly because you'll use them constantly.

The most important thing to understand about Linux on a Pi is that you will almost never use a graphical desktop. The Pi will sit on your network, headless, and you'll

interact with it entirely through a text terminal over SSH. This sounds primitive until you realize it's actually more convenient: you can manage every Pi in your house or clubhouse from a single laptop without ever leaving your chair.

SSH and PuTTY

What SSH Is

SSH stands for Secure Shell, and it's how you talk to a remote Linux machine. You type a command on your laptop, it executes on the Pi, and the output comes back to your screen. Every interaction in this book — every program installation, every configuration change, every debugging session — happens over SSH. It is also exactly the channel Claude Code uses to operate a Pi on your behalf: when Code reaches into a machine, it is doing it over SSH.

Installing PuTTY

On Windows, the standard SSH client is PuTTY. It's free, open-source, and has been the default Windows SSH client for decades. Download it from putty.org — grab the installer (the .msi file), not just the standalone executable, because the installer also includes related tools like PuTTYgen for SSH key generation and PSCP for file transfer. Run the installer and accept the defaults. PuTTY will appear in your Start menu.

On Mac or Linux, SSH is built into the terminal. You type ssh pi@192.168.1.50 and you're connected. No additional software needed. The PuTTY-specific instructions below still apply conceptually — the same settings need to be configured — but the interface is the command line rather than a graphical window.

Using PuTTY

When you open PuTTY, you see a configuration window. The most important field is "Host Name (or IP address)" at the top — enter the Pi's IP address, like 192.168.1.50. The port should be 22, which is the default for SSH. The connection type should be SSH.

Before you click Open, take a moment to save this as a named session. In the "Saved Sessions" field, type a name — "Bob", "GOES Pi", whatever helps you identify the machine — and click Save. From now on, you can double-click that name in the list to connect without retyping the address. When you manage multiple Pis, your PuTTY session list becomes your control panel.

PuTTY can also handle serial connections over USB, which matters when you're connecting a Pi directly to a radio via a serial cable. The IC-9700 CI-V interface, for example, uses a USB-to-serial connection. In PuTTY, change the connection type from SSH to Serial, enter the COM port number (visible in Windows Device Manager), and set the baud rate to match the radio's CI-V settings. Save this as a named session too — "IC-9700 CI-V" — so the settings are ready next time.

Essential Linux Commands

The following commands are the ones you'll use most often. This is not a comprehensive Linux reference — it's the subset that covers what you need for the projects in this book. If you need a command that isn't listed here, ask Claude — it knows every Linux command and can explain any of them in context.

Command

What It Does

ssh user@host

Connect to a remote machine. Replace user with the login name (usually pi) and host with the IP address.

ls

List files in the current directory. Add -la for detailed listing including hidden files.

cd /path/to/dir

Change to a different directory. cd ~ takes you home. cd .. goes up one level.

cat filename

Display the contents of a file. Useful for reading configuration files and log files.

nano filename

Open a simple text editor. Ctrl+O saves, Ctrl+X exits. Good for quick config edits.

sudo command

Run a command with administrator privileges. Required for installing software and editing system files.

sudo apt update

Refresh the list of available software packages. Run this before installing anything.

sudo apt install name

Install a software package. Example: sudo apt install python3-pip

pip install name

Install a Python package. On Pi OS, add --break-system-packages if prompted.

python3 script.py

Run a Python program. Claude Code writes Python; this is how you run it.

systemctl status name

Check whether a service is running. Example: systemctl status openclaw

sudo systemctl start name

Start a service. Replace start with stop, restart, enable, or disable.

journalctl -u name -f

Follow the live log output of a service. Ctrl+C to stop watching.

crontab -e

Edit scheduled tasks. Each line is a job that runs at a specified time.

ifconfig

Show network configuration including IP addresses. ip addr also works.

df -h

Show disk usage. Check how much space is left on the SD card or SSD.

htop

Interactive system monitor showing CPU, memory, and running processes. q to quit.

Running Python Programs

Many of the projects in this book involve Python programs written by Claude Code. You don't need to understand Python to use them. You need to know three things: how to run a Python program, how to install Python libraries that the program depends on, and how to stop a running program.

To run a program: python3 script.py. That's it. If the program needs command-line arguments, Claude Code will tell you what they are when it writes the program. To install a library: pip install libraryname --break-system-packages. The --break-system-packages flag is required on current Raspberry Pi OS because the system Python is managed by the OS package manager. It sounds alarming but it's fine for our purposes. To stop a running program: Ctrl+C. This sends an interrupt signal that terminates most programs cleanly.

Services: Programs That Run Forever

Most ham radio Pi applications aren't programs you run once and close. They're services that start when the Pi boots and run continuously until you tell them to stop. The SatNOGS ground station client, the WSPR transmitter, the OpenClaw AI agent, the Node-RED automation server — these all run as services.

Linux manages services through a system called systemd. The essential commands are straightforward. sudo systemctl start openclaw starts the OpenClaw service. Replace start with stop, restart, status, enable (to start automatically at boot), or disable (to prevent auto-start). When something goes wrong, journalctl -u openclaw -f follows the live log, showing exactly what the service is doing and what errors it's encountering. Claude Code can create systemd service files, install them, enable them, and troubleshoot them; Chapter 13 shows this in detail.

Scheduling: Cron Jobs

Sometimes you need a program to run at a specific time rather than continuously. A propagation report runs every three hours. A satellite pass prediction script might run once a day. A backup script might run weekly. The Linux scheduling tool is cron. You edit the cron table with crontab -e and add lines that specify when to

run a command. The format uses five fields — minute, hour, day of month, month, day of week — followed by the command. For example, the line "0 */3 * * * python3 /home/pi/propagation_report.py" runs the propagation report script at the top of every third hour. If you don't remember the cron format, ask Claude. It will write the crontab entry for you and explain what each field means.

Storage and Power

Every Pi boots from a micro SD card. This is one of the Pi's greatest strengths — swap cards to change what the Pi does — and its most common failure mode, because SD cards wear out under continuous write loads. For Pis that need to survive continuous writes, an external SSD or NVMe drive is the right answer: an M.2 SSD through a HAT on the Pi 4, or a native NVMe drive on the Pi 5. For lightweight, read-mostly applications — a WSPR transmitter, an APRS iGate — a good-quality SD card from a reputable brand (SanDisk, Samsung) is fine; keep a backup image so you can restore quickly if a card fails.

A Pi draws between 2 and 5 watts, so you can leave them running permanently. Use the official Raspberry Pi power supply or a known-good equivalent — USB-C at 5V/3A for the Pi 4 and 5, micro-USB at 5V/2.5A for the Pi 3 and Zero. Cheap phone chargers cause more Pi problems than any other single factor; they sag under load, causing random crashes, SD card corruption, and intermittent network failures that are extremely difficult to diagnose.

A Final Word: Let Code Do the Talking

Everything in this appendix is background, not a syllabus. You do not need to master it before you start. The whole point of the workflow in this book is that once a Pi is set up, you rarely touch it directly again — Claude Code does the talking. It logs in over SSH, runs the commands, reads the errors, installs the services, edits the configs, and keeps going until the job is done. The commands, the cron syntax, the systemd files, the Linux trivia — Code handles all of it.

So treat this appendix as orientation, not homework. Learn as much of it as you enjoy learning, and let Code carry the rest. You only need to participate at whatever level you want to. Some hams will want to understand every command; others will be happy to describe the goal and let Code do the work. Both are fine. The Pi is the hardware platform, Linux is the operating system, and Claude is the operator — and the operator is the part you can delegate.

Appendix B — Node-RED for Ham Radio

I discovered Node-RED several years ago and immediately understood why it exists. Most programming environments show you code — lines of text, functions calling functions, variables pointing at other variables. Node-RED shows you data flowing through a process. You see boxes connected by lines, and the lines represent the path that information takes from input to output. It is, in a very real sense, a visual representation of what your program does.

That visual quality is why Node-RED became my go-to programming environment for clubhouse automation before Claude Code came along, and why it's still used for applications where I want to see what's happening and make changes on the fly. The satellite rotator controllers, the APRS weather station interface, indoor temperature telemetry, and several other systems at the clubhouse are all programmed in Node-RED. When something isn't working, I can open the flow in a browser and watch the data move through it in real time.

This appendix explains what Node-RED is, how it's used in ham radio, and — most importantly for this book — how Claude Code interacts with it. That last point involves a surprise: Node-RED looks graphical to you, but underneath it's JavaScript, and Claude can read JavaScript just fine.

What Node-RED Is

Node-RED is a flow-based programming tool that runs in a web browser. It was originally developed by IBM for Internet of Things applications and is now an open-source project with a large community. You install it on a Raspberry Pi (or any Linux machine), open a browser to port 1880, and you see a canvas where you can drag, drop, and connect functional blocks called nodes.

A node is a single unit of processing. There are nodes that read data from an MQTT broker, nodes that parse JSON, nodes that make HTTP requests, nodes that write to a file, nodes that send email, and hundreds of others. You build a program by connecting nodes together into a flow: data enters at one end, gets processed through a chain of nodes, and exits at the other end. The connecting lines between nodes show exactly how data moves through the system.

The result is a program that's nearly self-documenting. When I look at the rotator controller flow, I can see at a glance that a button press triggers a message, which feeds into a function that calculates the target azimuth, which sends a command to the relay HAT, which activates the motor. If I need to understand why the rotator isn't moving, I can click on any connection line and see the actual data that flowed through it. This kind of visibility is difficult to achieve in a traditional text-based programming environment.

How Node-RED Is Used at the Clubhouse

Node-RED runs on several Pis at the W2MMD clubhouse. Each installation handles a different automation task, and most of them run continuously as services.

Satellite Rotator Control

The most complex Node-RED flow at the clubhouse is the satellite rotator controller. It reads target azimuth and elevation from a satellite tracking program, converts those to motor commands for the BEVRLink relay HAT, reads the current antenna position from an ADS1115 analog-to-digital converter that measures the rotator's position potentiometer, compares current position to target position, and activates the appropriate relays to rotate the antenna to the target. The flow handles clamping logic to prevent the rotator from exceeding its mechanical limits, deadband to prevent hunting around the target, and error detection for stuck or runaway conditions.

This is a nontrivial control system, and it's all visible in the flow editor. Every decision point, every calculation, every hardware interaction is a node on the canvas with lines showing how they connect. When the rotator wasn't responding to commands during a satellite pass, I could open the flow and see exactly where the data path was breaking.

APRS Weather Station

The weather station interface reads data from the station's serial output, parses the weather data (temperature, humidity, barometric pressure, wind speed and direction, rainfall), formats it into APRS packets, and feeds it into the APRS network. This is a straightforward linear flow: serial input, parsing, formatting, transmission. Node-RED handles it cleanly because each step is a discrete node that can be independently tested and monitored.

Other Applications

Additional Node-RED applications at the clubhouse include indoor temperature telemetry from multiple rooms, APRS packet processing and logging, and various data display dashboards that aggregate information from multiple sources into a browser-accessible overview. Each of these runs on its own Pi or shares a Pi with a related application.

What's Underneath: JavaScript

Here is the thing about Node-RED that matters for this book: it's not really a graphical programming language. It's a graphical front end for JavaScript. Every node, every connection, every flow is represented internally as a JSON file that describes the nodes and their JavaScript code. The graphical canvas is a convenience for humans. The actual program is text.

This distinction is irrelevant when you're building and editing flows in the Node-RED editor. But it becomes important when you want Claude Code to help you

understand, debug, or modify a flow. Claude can't see the graphical canvas. What it can see is the JavaScript underneath, and that turns out to be enough.

What Claude Code Can and Cannot Do with Node-RED

I stumbled onto this capability by accident. The satellite rotator controller had a bug — pressing the button to rotate to a specific azimuth did nothing under certain conditions. I could see in the flow editor that the message was leaving the button node, but it wasn't reaching the relay output. Something in the middle was eating it.

I had been using Claude Code for other projects, so on a whim I exported the flow's JSON file and gave it to Code. I asked it to trace what happens when the rotate button is pressed. Code read the JavaScript inside each function node, followed the logic through the message routing, and identified the problem: a conditional check in one of the intermediate functions was testing a variable that had been renamed in an earlier edit but not updated in this particular branch. The message was being silently dropped because the condition was evaluating a variable that no longer existed.

I had spent an hour staring at the graphical flow trying to find this bug. Code found it in about thirty seconds by reading the JavaScript. It didn't need to see the graphical layout. It just needed the logic, and the logic is in the code.

What Code Is Good At

Claude Code is genuinely useful for several Node-RED tasks. It can read an exported flow file and explain what each node does, how they connect, and what the overall flow accomplishes. It can trace the path of a specific message through the flow and identify where it's being modified, filtered, or dropped. It can find bugs in function nodes — the custom JavaScript blocks where most logic errors hide. And it can rewrite individual function nodes when you need a specific calculation, transformation, or conditional check that you don't know how to code in JavaScript.

This last point is particularly useful. Node-RED function nodes are where the custom logic lives — the code that's specific to your application rather than generic data routing. If you need a function that converts a raw ADC reading to degrees of azimuth, or one that calculates the Doppler shift for a satellite at a given elevation angle, or one that parses a non-standard serial protocol from a weather station, you can describe what the function should do and Claude will write the JavaScript for it. You paste the code into the function node in the Node-RED editor and it works.

What Code Is Bad At

Claude Code cannot create Node-RED flows. I've tried. The results are technically valid JSON that Node-RED will import, but the graphical layout is a mess. Nodes pile on top of each other, lines cross randomly, the spatial organization that makes

a Node-RED flow readable is completely absent. The flow works functionally but is unusable visually, which defeats the entire purpose of using Node-RED in the first place.

This isn't surprising when you think about it. Claude understands logic, data flow, and JavaScript. It does not understand spatial layout, visual aesthetics, or the conventions that make a Node-RED flow readable to a human. The graphical layout is metadata that's separate from the program logic, and Claude treats it as an afterthought because, to Claude, it is one.

The practical implication is clear: use Claude Code to understand, debug, and write function-level code within Node-RED. Do the graphical flow design yourself in the Node-RED editor. This division of labor works well. You handle the visual structure and the high-level data routing. Claude handles the JavaScript details inside the function nodes. Each party does what it's good at.

When to Use Node-RED vs. Claude Code

With Claude Code now available for writing Python programs and deploying them directly to Pis, a reasonable question is whether Node-RED is still worth using. The answer depends on what you're building and how you want to interact with it.

Node-RED is still the right choice when you want to see the data flowing in real time. If you're debugging a hardware interface — reading serial data from a weather station, controlling relays, monitoring ADC inputs — the ability to click on a connection line and see the actual data that just passed through it is invaluable. Node-RED's debug nodes and dashboard widgets give you live visibility into what's happening that a Python script running in a terminal simply doesn't provide without writing additional monitoring code.

Node-RED is also the right choice for flows that you expect to modify frequently. The graphical editor makes it easy to add a new branch, reroute data, or disable a section of the flow temporarily. You can make changes while the flow is running and deploy them with a single click. A Python program requires you to edit a file, save it, and restart the service.

Claude Code is the better choice for tasks that are primarily computational: processing an IQ file, generating a report from API data, writing and deploying a web server, automating a complex multi-step workflow. These tasks don't benefit from visual data flow because there isn't much to see — the data goes in, computation happens, results come out. Claude Code can write, deploy, and debug these programs faster than you could build equivalent flows in Node-RED.

In practice, the two coexist. At the clubhouse, Node-RED handles the hardware interfaces and real-time monitoring. Claude Code handles the computation, reporting, and infrastructure automation. They often connect to each other: a Node-RED flow might feed data to a Python script that Code wrote, or a Code-deployed web server might pull data from a Node-RED dashboard.

Getting Started with Node-RED

Node-RED is installed on a Pi with a single command: sudo apt install nodered. Once installed, you enable and start it as a service: sudo systemctl enable nodered followed by sudo systemctl start nodered. Open a browser on any computer on the same network and navigate to http://your-pi-ip:1880. You'll see the flow editor.

The Node-RED documentation at nodered.org is excellent, and the community library at flows.nodered.org has thousands of contributed nodes for everything from MQTT to database access to hardware interfaces. For ham radio specifically, there are nodes for serial port communication (for rig control and data interfaces), GPIO access (for relay control on the Pi), and UDP/TCP networking (for connecting to SDR software and network services).

If you're new to Node-RED, the best way to learn is to build a small flow: read a temperature sensor, display the value on a dashboard gauge, and log the readings to a file. This takes about twenty minutes and teaches you the core concepts of nodes, flows, messages, and deployment. After that, the rotator controller and weather station flows described in this book will make sense.

And if you get stuck — on the Node-RED side or the JavaScript side — export your flow and give it to Claude Code. It will read the JavaScript underneath and tell you exactly what's happening.