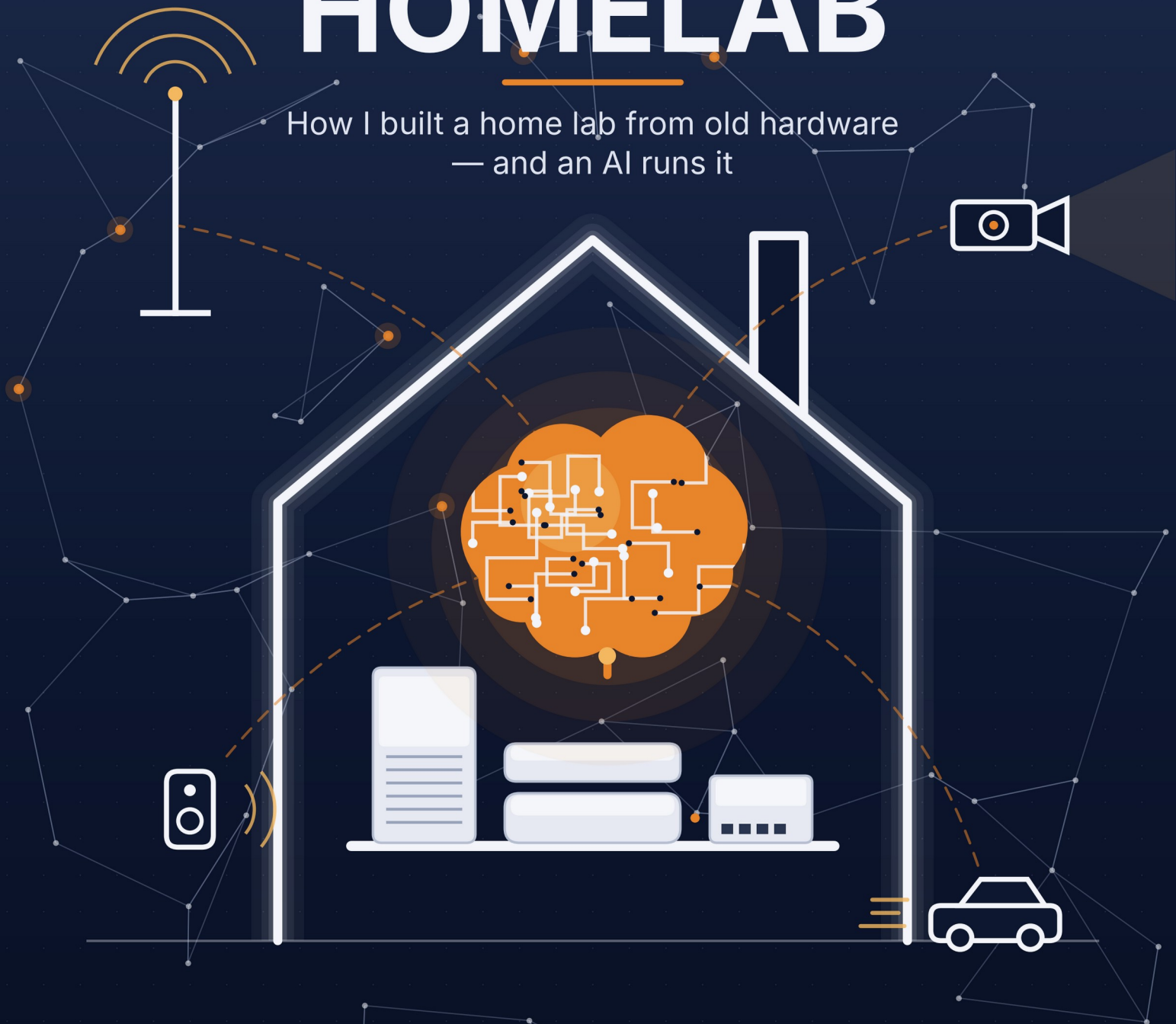


HAND-ME-DOWN HOMELAB

How I built a home lab from old hardware
— and an AI runs it



LUIS ALBISU

with an AI at the keyboard

Hand-Me-Down Homelab

How I built a home lab from old hardware — and an AI runs it

LUIS ALBISU

Hand-Me-Down Homelab: How I built a home lab from old hardware — and an AI runs it

Copyright © 2026 Luis Albisu. All rights reserved. No part of this book may be reproduced without the author’s permission, except for brief quotations in reviews and articles.

The companion files at handmedownhomelab.com/code (scripts, configuration files and playbooks) are licensed separately, under the MIT license, and you’re welcome to use them.

Product and company names in this book are trademarks or registered trademarks of their owners, used only to identify their products. Neither the author nor this book is affiliated with, sponsored by or endorsed by any of them.

This book describes the author’s own home lab as it was in October 2026. It is provided as is, without warranty of any kind, and is not legal, electrical or other professional advice. You follow it at your own risk; see “Your lab, your risk” in How to use this book.

Written with the help of AI assistants, chiefly Anthropic’s Claude, from the author’s own lab, notes and records. The author is responsible for its content.

Version 1.0 · October 2026

Corrections and updates: handmedownhomelab.com/code, in ERRATA.md

Contents

How to use this book.....	vi
A house run by old hardware and an AI.....	x
FOUNDATIONS.....	1
1 Hardware and network: old machines, hypervisors, backups and VLANs.....	2
1.1 The hardware pile: which old box is good for what.....	2
1.2 Proxmox on a 2012 Mac mini.....	6
1.3 A second Proxmox host.....	8
1.4 Backups that tell you if they fail.....	10
1.5 The gateway, and DHCP reservations as code.....	13
1.6 An IoT VLAN, and the camera VLAN I canceled.....	16
2 Give the lab a brain: the LLM wiki and an AI sysadmin.....	20
2.1 The LLM wiki: a lab that documents itself.....	20
2.2 One wiki on four machines: git hub, hooks, auto-sync.....	24
2.3 Claude Code as sysadmin: SSH, skills, memory, hooks.....	26
2.4 Teaching the AI from its mistakes: the memory file.....	30
2.5 The Homelab Menu and audit.sh: one command, sixteen hosts.....	32
3 Keeping it alive: monitoring, alerts that arrive, and an AI that reads them.....	36
3.1 Uptime Kuma: a monitor that is not one of the things it watches.....	36
3.2 Central syslog, and disk-health alerts.....	39
3.3 Email that actually arrives.....	40
3.4 The 06:32 triage: an AI reads the alerts before you do.....	42
3.5 Update sweeps without breaking things.....	44
THE SMART HOME.....	48
4 Home Assistant I: devices and automations.....	49
4.1 Home Assistant in a Proxmox VM, add-ons and MQTT.....	49
4.2 Zigbee buttons, door sensors, and a door chime spoken by the house.....	52
4.3 mmWave radar stair lights with ESPHome.....	54
4.4 WLED: garage and holiday porch lights.....	56
5 Home Assistant II: dashboards, the nightly report, and the AI.....	59
5.1 Dashboards I: the main home, energy and water.....	59
5.2 Dashboards II: servers, air and climate, TV and the family logbook.....	60
5.3 The nightly house-summary email.....	61
5.4 Sensors that lie: air quality, radon and stale values.....	63
5.5 Claude drives Home Assistant through MCP.....	65
YOUR NETWORK.....	69
6 Names, certificates and remote access.....	70

6.1 Pi-hole: ad blocking and local names.....	70
6.2 Two Pi-holes that stay in sync.....	74
6.3 NGINX Proxy Manager: one front door.....	77
6.4 Your own certificate authority: real HTTPS at home.....	80
6.5 Tailscale: your lab from anywhere, no open ports.....	84
YOUR OWN AI, AND APPS	87
7 The local AI box.....	88
7.1 Building the GPU box: Mint, NVIDIA, Docker and the upgrade traps.....	88
7.2 Ollama and Open WebUI: a private ChatGPT on your own GPU.....	91
7.3 Forge and FLUX: images by chatting with Claude.....	94
7.4 ComfyUI: text and photos to video.....	96
7.5 Fix old photos by typing: FLUX Kontext in Open WebUI.....	98
7.6 ACE-Step: whole songs on your own GPU.....	99
7.7 One 16 GB GPU, seven AI apps: who gets the card?.....	101
8 Agents and your own apps.....	104
8.1 Hermes: a sandboxed AI agent on local models.....	104
8.2 Texting the house, safely.....	106
8.3 Claude from your phone: Remote Control on an always-on Mac mini.....	107
8.4 Photo Intel I: getting 228,000 photos out of Apple Photos.....	108
8.5 Photo Intel II: descriptions, places, text and faces.....	109
8.6 Photo Intel III: the search app, and finding duplicates.....	111
8.7 HomelabMenu: an iPhone app when you're not an iOS developer.....	112
9 Files, movies and a radio.....	114
9.1 Nextcloud: private file sync with no open ports.....	114
9.2 Home movies: Whisper transcription and Plex.....	115
9.3 OpenWebRX: a web radio receiver on a Pi Zero.....	116
SECURITY	119
10 Eyes on the street: cameras and speed.....	120
10.1 A video recorder plus Frigate: recording and detection.....	120
10.2 Blind for 45 hours: monitoring the work, not the worker.....	122
10.3 Measuring speeders on the street.....	125
10.4 A portable speed camera: the build, and the plan to calibrate it.....	126
11 Watching the wire: intrusion detection and Wi-Fi.....	131
11.1 LanScan: what just joined my network?.....	131
11.2 Suricata and EveBox on a Raspberry Pi 5.....	132
11.3 Tuning, and the daily AI digest.....	134
11.4 Kismet: passive Wi-Fi monitoring.....	137
11.5 Watching the cable modem.....	139
11.6 A security audit run from the wiki.....	140
What it cost, what broke, and what I'd do differently.....	143
A The minimum viable lab.....	147
B Redaction and privacy for people who publish their homelab.....	151

C The AI starter kit.....	154
D Glossary.....	160
E Software and where to get it.....	164

How to use this book

Who this book is for

You're comfortable in a terminal. You've used SSH, you know roughly what an IP address and a DNS server are, and you have an old computer in a drawer, or a Raspberry Pi you bought and never quite found a use for. You'd like your house to be smarter and your own data to stay yours, and you've wondered whether an AI could do some of the heavy lifting.

You don't need to have used a virtual machine manager like Proxmox, written a Home Assistant automation or trained a model. Each chapter explains what it uses the first time it uses it, and the Introduction starts with a short tour of the tools. If you're more experienced, skip the explanations and read the war stories: that's where the value is for you.

What's different about this book

Plenty of good homelab guides exist. This one has three things most don't:

1. **Genuinely old hardware.** A 2012 Mac mini runs my most important services, and the biggest machine is built from a friend's spare parts. Its graphics card is the one computer part I bought new.
2. **Real failures, with dates.** Every chapter includes war stories from my own lab, told as *symptom* → *wrong guess* → *cause* → *fix* → *lesson*. Most guides show you the path that works. The lessons are in the paths that didn't.
3. **An AI doing the work.** Nearly everything in this book was built by me and an AI assistant together, and Chapter 2 shows you how to set that up so it's safe and actually useful.

How the chapters work

Every chapter says what it needs at the top, in a box like this one:

You'll build: what the chapter produces.
You'll need: hardware and accounts.
Difficulty: and roughly how long it takes.

And every chapter ends with a short checklist, **Before you move on**, so you know you're done before you go further.

You don't need all of it. One old PC and one Raspberry Pi cover most of the book. **Appendix A** goes through it chapter by chapter, and the Introduction has the map of the whole book.

The boxes

Box	What it's for
War story	Something that broke in my lab: symptom, wrong guess, cause, fix, lesson

Box	What it's for
AI at the keyboard	What the AI did, what to ask it, and where a human was needed
Tip	A shortcut or better way
Watch out	A mistake that costs you data, security or a weekend
Note	Background worth knowing
Where this stands	Work that's built but not yet finished, and what's left

Commands, addresses and names

- Commands are shown as you'd type them. When a command runs on a particular machine, the text says which.
- The addresses in examples (10.20.x.x) aren't my real ones. I swapped them for made-up private addresses, consistently, so a machine has the same address everywhere in the book and the companion files. Use your own.
- **Tom is the fictional user.** Logins and home folders appear as `tom`, as in `tom@lmstudio.home.arpa` or `/home/tom`. Tom is a stand-in for me at the keyboard; wherever you see `tom`, put your own username.
- Machine names end in `.home.arpa`, the name space reserved for home networks (Chapter 6 explains). Two names come up often enough to introduce now: **lmstudio** is the GPU tower, the biggest machine in the house; **minim2** is the Mac mini M2 that's always on and runs the AI from the loft.
- The longer scripts are in the book's companion files, free to use under the MIT license: **handmedownhomelab.com/code**. Each section names its file by section number, such as `6.2/pihole-sync.sh`.
- **Installs are playbooks, not pages of commands.** Where a section installs something, it opens with a **Get it** line: the project's own website and the playbook's short address, such as **handmedownhomelab.com/p/6.1-pihole**. The chapter tells you what I did and why. The playbook is written for your AI assistant to carry out on *your* network: it finds your addresses instead of using mine, takes current install steps from the project's own documentation, asks before anything risky, and checks each step by its effect. Each one was tested on a clean machine before it was published, except the few that need hardware a test machine can't have, such as a graphics card: those say **Not tested** at the top, and why. Every playbook reads fine as a checklist if you'd rather type the commands yourself.
- **Read a playbook before you hand it over.** It's instructions an AI will carry out with your permissions, so treat it the way the book treats an installer script: look first. Download it, read it, and give your AI that copy, not the web address, so the file you read is the file that runs.
- Every program the book uses, with its website, is listed in **Appendix E**.

- Anything that would identify my family, my street or my accounts has been left out or re-placed. Appendix B explains how, so you can do the same when you publish your own lab.

About the AI

The examples use **Claude Code**, Anthropic's command-line AI assistant, because that's what I use. The ideas (a written memory the AI maintains, house rules it reads, a record of its own mistakes, narrow tools) apply to any capable assistant. Where a specific product matters, the text says so.

An AI assistant makes mistakes. So do I. The book is honest about both, and the systems it describes are designed around the assumption that mistakes will happen: things are checked by reading the effect, risky actions need a human's yes, and every lesson is written down.

How this book was made

I built this lab and I run it. Every machine, every choice of software and every mistake in these pages is mine, made in my house over the last few years. The lab keeps its own records: a wiki of every machine and service, a log of every change, and the notes the AI keeps about its own mistakes. Those records are where this book comes from.

The writing is where the AI comes in. Claude drafted the chapters from those records, and a later Claude model revised the whole book. I read every draft, corrected what it got wrong, and decided what stayed. Then each chapter was checked against the running lab in October 2026: the commands, the versions, the numbers. Where an install couldn't be re-run because it needs hardware I'd have to take down, its playbook says **Not tested**. Other AI models and a human reader reviewed it too, and their corrections are in.

It's the same way I run the house. The AI does a lot of the work, I make the decisions, and nothing counts as done until it's been checked against the real thing. If you find something wrong, it goes in the errata (handmedownhomelab.com/code).

Versions and prices

Everything here was true in October 2026. Software moves quickly, and AI software moves fastest. Version numbers are given where they matter. When something doesn't match what you see, trust what you see, and check the project's own documentation.

Your lab, your risk

- This book describes what I did, on my equipment, in my house. Doing the same on yours is your decision and your risk. **Back up first**, and try anything new on a spare machine or a throwaway virtual machine before it touches something you rely on.
- **Nothing here is legal, electrical or other professional advice.** Where the law matters (recording a street, monitoring Wi-Fi, listening to radio), the text says so and asks you to check the rules where you live. Nothing in this book is wired to mains power; if a project of yours is, that's an electrician's job. The low-voltage builds still need a fuse on every power run (4.4).
- **The book and the companion files come with no warranty.** I tested what I could, and say plainly where I couldn't.

- **Other people's privacy isn't yours to publish.** Your cameras, your Wi-Fi monitor and your radio all pick up neighbors. Appendix B is how I keep them out of everything I share.

A house run by old hardware and an AI

At 6:32 every morning, an AI reads the alert emails my house sent overnight and decides whether any of them need me. Most mornings, none do. At 11:50 every night, the house emails me a report card: energy, water, air quality, the health of every server, and a banner at the top that stays green unless something is wrong. In between, it announces which door just opened in a voice generated on my own hardware, turns the stair lights on when radar says someone's on the stairs, measures the speed of every car that passes, watches the network for intruders, and answers my questions about 228,000 family photos.

Nearly all of that runs in my basement, on computers that other people were done with. The exception is the assistant that reads the morning email: that's Claude, a subscription service, and Chapter 2 shows what it's allowed to touch.

This book is how I built it, and how you can build your own version, starting from whatever old machine you have in a drawer. Many homes have one like that, doing nothing. It can still do useful jobs for your home, and today an AI assistant can do most of the setup with you. Building it that way also turns out to be one of the best ways I know to learn how to use AI.

Two words first

A **server** is just a computer that stays on and does a job for the other devices in your home. It doesn't need a screen or a keyboard. It sits on a shelf and works.

A **homelab** is a small collection of servers that you set up at home yourself. It's called a lab because you experiment and learn as you go.

Every other term is explained the first time it's used, and Appendix D is a glossary of the ones that come up most.

What a homelab can do

The jobs fall into five groups, and after the foundations, each group is a part of this book.

- **The smart home** (Part II): one app for lights, locks, sensors and plugs from many different brands, all working together. It's where I started.
- **Your network** (Part III): ads blocked on every phone, tablet and TV in the house at once, and a simple name for every device.
- **Your own AI, and your own apps** (Part IV): a private chatbot on your own computer, where nothing you type leaves the house; and the things my family actually notices, like private file sync, searchable home movies and a web radio.
- **Security** (Part V): an alarm system for your network, and cameras smart enough to tell a person from a car.

And underneath all of it, **Part I**: the hardware, the network, the notes, the AI, and monitoring that watches everything you build from the day you build it.

Almost all of this software is free. The hard part has never been the cost; it's been knowing how to set it up. That's where the AI comes in.

The pile

Almost everything here is a hand-me-down, which is where the book's title comes from.

- **A 2012 Mac mini.** I bought it in 2012. It changed hands in the family over the years, and I got it back. It's now the most important computer in the house: it runs the smart home, the DNS for every device, the reverse proxy and my own certificate authority.
- **A desktop tower.** The motherboard, processor and memory came from a friend who wasn't using them; the motherboard dates from 2014. The case is a family hand-me-down. The graphics card is the only part of it I bought new, and that one card now runs the camera AI, a private chatbot, and makes images, video and music. On the network it's called **lmstudio**, and you'll see that name in commands throughout the book.
- **A 2009 Mac mini Server**, from a friend. Seventeen years old. An SSD made it usable again, and it has no job yet, on purpose.
- **A tiny N100 mini PC and two Raspberry Pis**, bought for very little: a second hypervisor, a network intrusion sensor and a web radio receiver. A third Pi, an old Pi 3 from a drawer, is on the bench becoming a portable speed camera.
- **A Mac mini M2**, the one computer here that was neither handed down nor bought for the lab. It was my desktop, and because it's always on, it became the machine the AI works from. On the network it's **minim2**.

What you need

You don't need anything new or powerful. These jobs are light, and a computer from ten years ago can handle several of them at once.

The rule of thumb: if it has **8 GB of memory and a solid-state drive (SSD)**, it will do. If it doesn't have an SSD, that's the one upgrade worth buying. The processor almost never matters. An old gaming PC is a bonus, because its graphics card can run AI.

A few things won't work as the lab's server: very old 32-bit computers, anything with less than 4 GB of memory, and Apple's M-series Macs, because the software in this book needs an Intel or AMD processor. (An M-series Mac makes a fine workstation to run the AI from; mine does.)

If you don't have anything at all, a new mini PC with an Intel N100 processor started at around \$200 when I wrote this. A used office mini PC, like a Dell OptiPlex Micro or an HP EliteDesk Mini, often costs less, sometimes under \$100. And a Raspberry Pi makes a good second machine for a single job.

Running one small machine costs a few dollars a month in electricity. Bigger hardware costs more, especially a graphics card for AI; the Afterword has my real numbers.

Chapter 1 goes through which box is good for what, and Appendix A shows the smallest useful lab: one old PC and one Raspberry Pi.

Why an AI

I got interested in home automation, and found I could use an AI to help me set it up. It went on from there: one project led to the next, and before long more was running in this house than I could keep in my head. What I needed was a way to *remember* it all.

Now an AI remembers for me. Every time the AI changes something (and most of the configuration in this house was written by an AI at my direction), it updates the page for that machine in my wiki and writes a line in a log saying what changed and why. The wiki is organized like a library: a catalog at the front, a page for each computer and each service, and a log of every change. It now has about 125 pages, more than 1,400 log entries and 24 write-ups of things that broke. The next time the AI starts work, it reads what it wrote last time. That's why it remembers.

Those notes are also what let a small lab grow. Each new piece, whether a second computer, a graphics card for AI, or an alarm for the network, starts from everything the AI already knows about the lab, instead of from scratch.

It also remembers its mistakes. About ninety of them are now rules it reads before it starts, each one written as "before you do X, check Y". Chapter 2 is about that, and it's the chapter I'd most like you to read.

The AI runs the house day to day, but inside limits I set. The big decisions are still mine. It asks before anything irreversible. It can't touch money or passwords. And it's wrong often enough that the whole system is built around checking what it did. What it gives me is a tireless, well-documented assistant who can SSH into sixteen hosts (the eight boxes above, plus the virtual machines and containers they run, and a laptop), read a week of logs in a minute, and write down everything it learns.

Three ways to use it

You don't have to hand an AI the keys on day one. I'd suggest using the AI in three steps.

1. **Ask.** Describe your old computer and ask what it's good for. Ask the meaning of a technical word or phrase. Ask it to explain its plan. This costs nothing, and it can't break anything.
2. **Let it guide you.** Claude writes the commands, and you type or paste them in yourself. If something goes wrong, show Claude the message; it explains what happened and what to try next.
3. **Let it do the work.** If you feel you can trust it, let it do the work for you. Claude Code connects to your computers and makes the changes itself, inside rules you set, like the ones above. Claude has safeguards built in to keep it from doing things it shouldn't, but approach this step carefully.

When I started, I did the copying and pasting myself, so I knew what Claude was doing. Eventually I got comfortable enough to let Claude do the work while I watched. Do whatever feels comfortable to you. Either way works.

Steps one and two work with the free version of Claude. Step three needs a paid plan. I use Claude Pro, which was \$20 a month when I wrote this. Every plan has a usage limit, and because Claude reads its notes instead of working everything out again from scratch, it uses less. Pro has been enough for everything in this book.

The shape of it

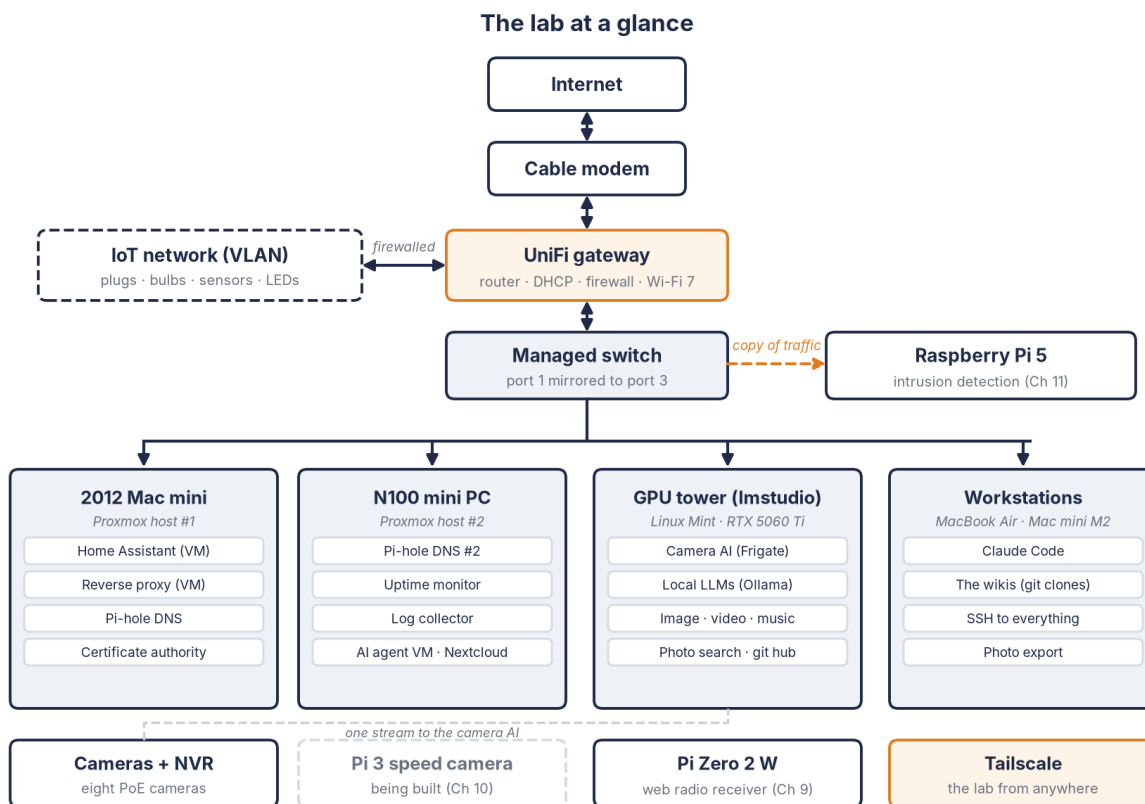


Figure I.1 · *The whole lab. Each chapter builds one part of this.*

At the top, the internet, a cable modem and a UniFi gateway. A managed switch copies the gateway's traffic to the intrusion sensor. Smart plugs, bulbs and sensors live on their own network, fenced off by the gateway's firewall. Four groups of machines do the real work: the two Proxmox hosts running the core services, the GPU tower running everything that needs AI, and the workstations where I, and the AI, sit. Tailscale lets me reach all of it from anywhere, without opening a single port to the internet.

The tools, and how I found them

When I started, I didn't know most of these existed. I found them the same way, over and over: I described a problem to Claude ("I'm tired of clicking through certificate warnings", "I want to know when something goes down, even when Home Assistant is the thing that's down") and asked what people use for it, what each option would need from my hardware, and which one it would pick. It laid out the options and I chose. The reasoning went into the wiki with the decision, so it survived: the gateway page still lists the models I ruled out and why, and the monitoring page says why alerts come by email and not by text message.

Here's the short tour. Each tool gets a proper introduction in the chapter where it's first used.

Tool	What it does here	Chapter
Proxmox VE	Runs virtual machines and containers, so one old PC can do several jobs	1
UniFi gateway	Router, firewall and separate networks (VLANs) for untrusted devices	1
Claude Code	The AI assistant: reads the wiki, logs in to machines, changes and documents things	2
Obsidian and git	Read the wiki, and keep a copy of it on every workstation	2
Uptime Kuma	Watches every service, including Home Assistant, and emails when one stops	3
rsyslog	Collects every machine's logs in one place	3
Pi-hole	Blocks ads for the whole house and gives local machines names	4
NGINX Proxy Manager	One front door for every web page the lab serves	4
Step-CA	My own certificate authority, for real HTTPS at home	4
Tailscale	Reach the lab from anywhere without opening a port to the internet	4
Home Assistant	The smart home: devices, automations, dashboards, the nightly email	5, 6
ESPHome, Zigbee, MQTT	How sensors, buttons and small boards talk to Home Assistant	5
Ollama and Open WebUI	A private chatbot running on my own graphics card	7
Forge, ComfyUI, ACE-Step	Images, video and music, on the same card	7
Nextcloud	Private file sync, like iCloud or Dropbox, on my own server	9
Frigate	Camera AI: spots people and cars in the video feeds	10
Suricata and EveBox	Intrusion detection: flags suspicious traffic on the network	11
Kismet and OpenWebRX	Listen to Wi-Fi and radio around the house	11, 9

Everything on that list is free for home use except Claude, which is a subscription.

AI AT THE KEYBOARD · TRY ASKING

Before you install anything, describe the problem, not the product: *“I want to [block ads on every device / see my lab from my phone / know when a service stops]. What open-source tools do people use for this? For each, tell me what it needs from this hardware: [your machines]. Which would you pick, and why? Then write the choice and the runners-up into the wiki.”*

One morning, in brief

A real Friday in September: the 6:32 triage email had flagged one item as needing action and six to keep an eye on. At nine I opened a session and handed it the email. Thirteen minutes later, all seven were settled. The one “action” item turned out to be the AI’s own test alert from the day before, which it found by reading its own log. One fix was blocked by its permission rules, so it handed me the command instead of looking for a way around. It treated a delete as mine to make, and it wrote everything down, including its own miscount.

That’s the book in miniature, and the full transcript is in Chapter 3, where the triage is built.

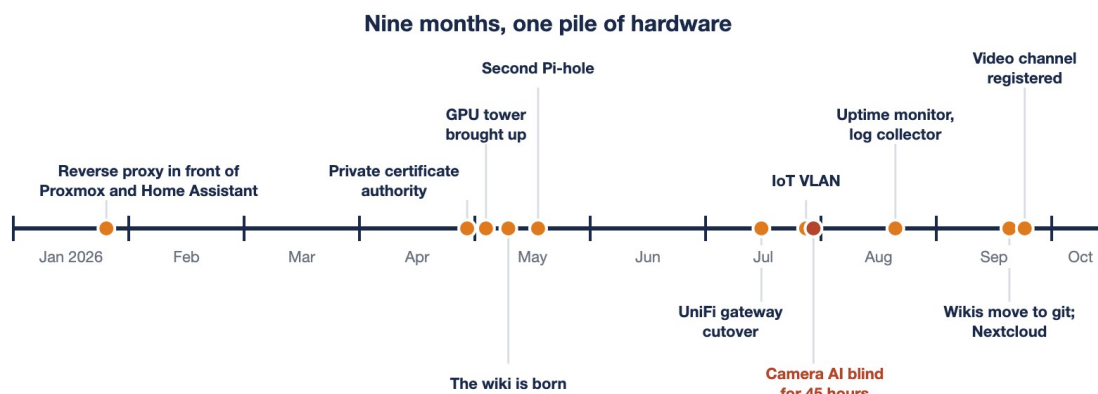
How it happened

Figure I.2 · The journey, from a reverse proxy to a video channel.

It wasn’t planned as a whole. Proxmox and Home Assistant were already running in January, when a reverse proxy went in front of them. A private certificate authority in April, because I was tired of clicking through browser warnings. The GPU tower in May, and the week it took to make a brand-new graphics card work on Linux. Then, on 10 May, the day that changed how everything else went: I watched a video explaining Andrej Karpathy’s idea of an AI-maintained wiki (2.1 has the credit and the link) and started one. Every date after that is in the log.

July was the network: a new gateway, a separate network for the smart-home devices, and the intrusion sensor moved to where it could see which device was doing what. July also brought the outage that taught me the most: the camera AI processed nothing for 45 hours while every status light said healthy. I found it because a traffic counter said zero cars, and that looked wrong. That one’s in Chapter 10, and it changed how I monitor everything.

In August, independent monitoring and a log collector. In September, the wikis moved off iCloud into git, private file sync arrived, and the idea for this book and a video series about it.

How the book is organized

The book is in five parts. **Read Part I first:** everything else stands on it. It's the hardware and the network (Chapter 1), the notes and the AI assistant (Chapter 2), and the monitoring that watches everything you build afterwards (Chapter 3). I built monitoring near the end of my own nine months and regret it; here it comes first, so that everything you add is watched from day one.

After Part I, go straight to whichever part you care about. Each part opens by saying what it needs from the others.

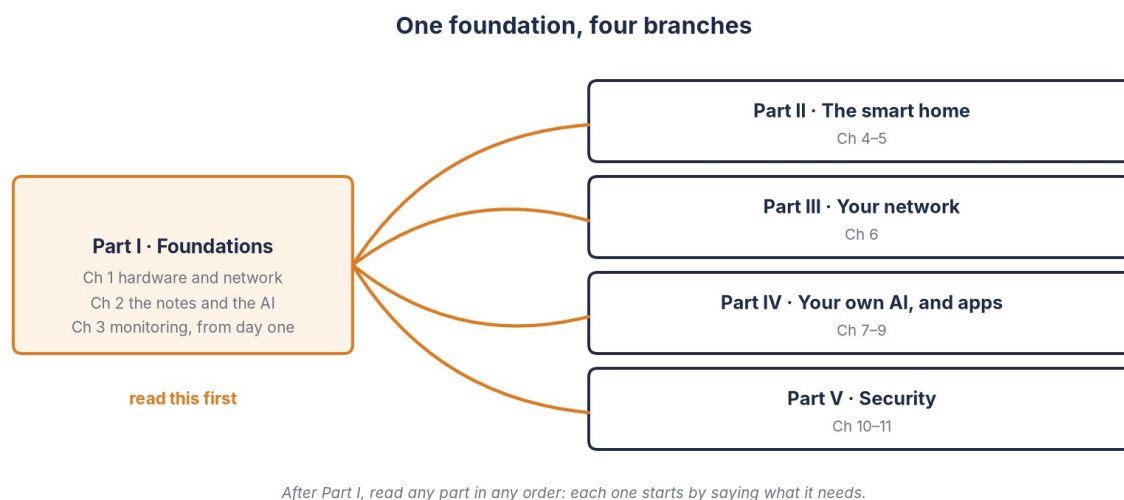


Figure I.3 · *One foundation, four branches: the book's five parts.*

Part	Chapters	You'll build	Needs
I Foundations	1 Hardware and network · 2 The brain · 3 Keeping it alive	Hypervisors, backups, the gateway, VLANs; a self-maintaining wiki and an AI sysadmin; monitoring, alerts that arrive, and an AI that reads them	nothing
II The smart home	4 Devices and automations · 5 Dashboards, reports and the AI	The smart home, its dashboards and its nightly report card	I (III helps)
III Your network	6 Names, certificates and remote access	Ad blocking and local names, a reverse proxy, HTTPS, remote access	I

Part	Chapters	You'll build	Needs
IV Your own AI, and apps	7 The local AI box · 8 Agents and apps · 9 Files, movies and a radio	A GPU server for chat, images, video and music; an AI agent, photo search, an iPhone app; file sync, home movies, a web radio	I and a GPU; II and III for some sections
V Security	10 Eyes on the street · 11 Watching the wire	Camera AI and speed measurement; new-device scanning, intrusion detection, Wi-Fi monitoring	I; the cameras need Chapter 7 and Part II

Each chapter says what you need, builds something that works, tells you what broke when I built it, and ends with a checklist. You'll get the exact commands, configuration and scripts. You'll also get the reasoning behind each decision, including the decisions to *not* build something, because a few of the best lessons are in those.

What you won't find: anything that identifies my family, my street or my accounts, and any security weakness I haven't fixed yet. Appendix B explains why and how, because if you document your lab the way this book suggests, you'll face the same questions when you share it.

If you've got an old computer in a drawer, you already have the first piece. If you don't, a small one costs less than you'd think. Either way, Chapter 1 starts with the question of what it's good for.

Foundations

Every other part of this book stands on these three chapters. Chapter 1 turns a pile of old computers into a lab: a hypervisor, backups that tell you if they fail, and a network with room to grow. Chapter 2 gives that lab its notes and its AI assistant. Chapter 3 gives it a monitor that is independent of everything it watches, alerts that have been proved to arrive, and an AI that reads them before you do.

Read all three before you jump anywhere else. After that, pick the part you care about; each one starts by saying what it needs.

1 Hardware and network: old machines, hypervisors, backups and VLANs

You'll build: two Proxmox hosts with backups that fail loudly; a gateway whose address reservations live in a text file; and a separate network for smart-home devices.

You'll need: one old PC or Mac for Proxmox (a second small machine for 1.3); a USB drive for backups; for 1.5–1.6, a gateway with VLANs and an API.

Difficulty: easy to moderate. **Time:** a couple of weekends, one section at a time.

Everything else in this book stands on the six things in this chapter. By the end of this one you'll have what I have in my basement: a pile of old machines sorted by what each is actually good for, two Proxmox hosts with backups that fail loudly instead of quietly, a gateway whose DHCP reservations live in a text file, and a separate network for the smart-home junk. Chapter 2 then gives the lab somewhere to write things down, and Chapter 3 gives it a monitor. After that comes the smart home (Chapters 4 and 5), and Chapter 6 adds names, certificates and remote access.

None of it is exotic. What's different is that every step here was done by me with an AI at the keyboard beside me, and most of it broke at least once. The breaks are in here too, because they're where the lessons are. If this chapter has a theme, it's *silent failure*: a mount that reported success and mounted nothing, backups that went somewhere nobody looked for 13 days, a firewall test that passed only because of where I ran it from. The refrain you'll hear for the rest of the book starts here: **verify the effect, not the exit code.**

1.1 The hardware pile: which old box is good for what

Here's the pile. A 2012 Mac mini. A 2009 Mac mini Server. A no-name N100 mini PC. Three Raspberry Pis of three different generations, two of them in service and one still on the bench. And one desktop tower built from a friend's spare motherboard, processor and memory, in a case that's a family hand-me-down.

Before you install anything, you need to know which box is good for what, and the answer is almost never the CPU.

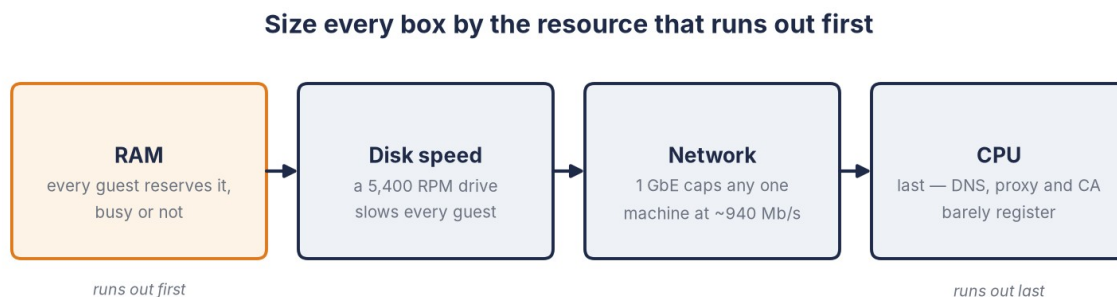
Size by the binding constraint

Every machine has one resource that runs out first. That's its *binding constraint*, and it's the only number that matters when you're deciding what it should run. For a homelab host the order is nearly always:

1. **RAM.** Every virtual machine and container reserves memory whether it's busy or not. Home Assistant alone wants several gigabytes. A host with 16 GB runs four or five services comfortably; one with 8 GB runs one or two.
2. **Disk.** Not capacity, *speed*. A 5,400 RPM laptop drive makes every guest on it feel slow. An SSD fixes that, but only up to the speed of the controller it's plugged into.

3. **Network.** Every host in this house has a 1 Gb Ethernet port, which caps any single machine at about 940 Mb/s. For most services that never matters.
4. **CPU.** Last. A home DNS server, a reverse proxy and a certificate authority together barely register on a twelve-year-old processor.

The exception is AI work, where the binding constraint is the graphics card's memory (VRAM). That's Chapter 7's problem.



The exception: AI work, where graphics-card memory (VRAM) is the constraint — Chapter 7.

Figure 1.1 · *The order in which a homelab host runs out of things.*

No Linux on it yet? Boot it from a USB stick

Get it: Linux Mint, linuxmint.com · balenaEtcher, etcher.balena.io · **Playbook:** handmedown-homelab.com/p/1.1-live-usb

The commands below run on Linux. If the machine runs Windows, or it's an Intel Mac on macOS, you don't have to wipe it to find out what it's worth. Boot it from a USB stick and appraise it from there: a *live* session runs Linux from the stick and leaves the internal disk alone.

1. Download Linux Mint (the Cinnamon edition) from linuxmint.com, and check the file against the SHA256 sum on the download page.
2. Write it to a USB stick of 8 GB or more with balenaEtcher, which runs on Windows, macOS and Linux. Writing erases the stick.
3. Plug the stick into the old computer and press its boot menu key as it powers on: usually F12, F11 or Esc on a PC (the playbook lists them by brand). On an Intel Mac, hold **Option**. Pick the stick, then *Start Linux Mint*.

If it won't start from the stick, the usual causes are a firmware setting, Secure Boot or USB boot switched off, and the playbook covers both. Apple Silicon Macs can't run it at all. The same skill, writing an image to a stick and booting from it, is how Proxmox goes on in 1.2.

Four commands

Four commands tell you what you're working with on any Linux box: `lscpu` for cores, threads and the virtualization flags; `free -h` for memory; `lsblk` for every disk (ROTA=1 means spinning); and `smartctl` for disk health and power-on hours.

```
lscpu
free -h
lsblk -o NAME,MODEL,SIZE,ROTA
sudo smartctl -a /dev/sda
```

If `smartctl` isn't found, install it with `sudo apt install smartmontools`. In a live session it lasts until you shut down.

Look for `vmx` (Intel) or `svm` (AMD) in `lscpu`'s flags. If neither appears, check the firmware settings before you give up: many old PCs ship with virtualization switched off, and Linux hides the Intel flag until it's on. The setting is usually called *Intel Virtualization Technology* (or VT-x) or *SVM Mode*. Only a processor that has no such setting at all can't run virtual machines.

AI AT THE KEYBOARD · TRY ASKING

Give the AI SSH access to the machine and ask: *"Inventory this box: CPU and virtualization flags, RAM, every disk with its model, type and power-on hours, network ports and their speeds. Then tell me what its binding constraint is and what it's good for."* Save the answer as the machine's first wiki page (Chapter 2). From a live session, install the SSH server first (`sudo apt install openssh-server`) and give the live user a password with `passwd`.

What each box ended up doing

Machine	Name	Specs that matter	Job	Why
2012 Mac mini	proxmox	16 GB RAM, 1 GbE, no PCIe slots, no VT-d	Proxmox host for the core services	Enough RAM for four guests; quiet and small
N100 mini PC	pmx2	16 GB RAM, 4 efficiency cores, 1 GbE	Second Proxmox host	Low power and always on; carries the services that must survive the first host going down
2009 Mac mini Server	old-macmini	Core 2 Duo (2 cores), 8 GB RAM, SATA II	Nothing, on purpose	See below
Raspberry Pi 5 + NVMe	ids	4 cores, fast storage, a USB Ethernet adapter for a second port	Network intrusion detection (Chapter 11)	Cheap, low power, one job
Raspberry Pi Zero 2 W	openwe-brx	512 MB RAM	Web radio receiver (Chapter 9)	Tiny, and radio reception needs almost no CPU
Raspberry Pi 3B + PS3 Eye camera	speedpi	1 GB RAM, USB 2	Portable speed camera (Chapter 10), built but not yet deployed	Will go out to the street on a tripod

Machine	Name	Specs that matter	Job	Why
GPU tower	lmstudio	i7-4790K, 32 GB DDR3, RTX 5060 Ti 16 GB	Cameras and local AI (Chapters 7, 8, 9 and 10)	The one thing worth buying new for is a GPU
Mac mini M2	minim2	Apple silicon, always on	The AI's workstation in the loft; Remote Control (8.3); photo export (8.4)	Can't run the lab's software, but runs the AI that does

The basement machines (both Proxmox hosts, the GPU tower, the intrusion sensor and the video recorder), with the modem, the gateway and the core switches, share one 800 VA battery backup (UPS), so a short power blip doesn't take the house's DNS with it. The Mac mini in the loft isn't on it.

What does it cost to run? The whole basement room (the two Proxmox hosts, the GPU tower, the intrusion sensor and the video recorder, plus the modem, gateway and switches) averages about 180 to 225 W, or roughly \$20 to \$26 a month at my electricity rate. One metering smart plug in front of all of it does the measuring; the Afterword has the numbers.

The hidden limits of old hardware

Old machines rarely fail on cores. They fail on the things nobody lists on the box:

- **No GPU passthrough.** Passing a graphics card through to a virtual machine needs PCIe slots and a CPU feature called VT-d. The 2012 Mac mini has neither, so it will never host AI or camera detection. That work goes to the tower.
- **Slow disk controllers.** The 2009 Mac mini's controller is SATA II. More on that in the war story at the end of this section.
- **Fans that don't work under Linux.** A Mac's fan curve is run by a chip called the SMC, and it takes its cues from macOS. Under Linux it gets no input and does something unhelpful. On the 2009 mini it pinned the fan at full speed (5,500 RPM, all day). On the 2012 mini it did the opposite: it parked the fan at its 1,800 RPM minimum and never moved it. Same root cause, opposite symptom, so don't assume one Mac tells you about another. The fix is in 1.2.

The box with no job

The 2009 Mac mini Server has no workload, and that's a decision, not a to-do. It sat on my list as "decide what this box is for" for a month, until I realized the item was backwards: it was an invitation to invent work for the hardware. A machine with nothing to do is a fact, not a defect. The constraints are written down for the day a real candidate turns up, and until then it idles. Not every old box needs a job.

The minimum viable lab

You don't need this pile. One old PC plus one Raspberry Pi covers most of the book: Proxmox on the PC runs the whole of this chapter apart from the second host, all of Chapters 2, 3 and 6 apart from the second Pi-hole, Home Assistant in Chapters 4 and 5, and the monitoring in Chapter 3 if you put it on the Pi. Appendix A goes chapter by chapter.

WAR STORY · FIXING ONE BOTTLENECK FINDS THE NEXT

Symptom. The 2009 Mac mini was painfully slow at everything: installs, updates, even logging in.

Wrong guess. Old CPU. A Core 2 Duo from 2009 seemed the obvious culprit.

Cause. The two original 5,400 RPM laptop drives. The disk was the binding constraint, not the processor.

Fix. Both drives came out and a single 1 TB SSD went in. It helped enormously, but the link negotiated at 3.0 Gb/s, not the 6.0 the SSD advertises. That's not a bad cable: the 2009 controller is SATA II, which caps the drive at roughly 275–300 MB/s. That's still several times what the spinners managed.

Lesson. With the disk fixed, the two CPU cores and the 8 GB of RAM became the new ceiling. Fixing a bottleneck doesn't make a machine fast; it shows you the next bottleneck. Measure which one you have before you buy anything.

1.2 Proxmox on a 2012 Mac mini

This Mac mini is older than some of the people who'll read this book, and it runs four of the most important services in the house, each in its own box.

Guest	Type	Job
Home Assistant	Virtual machine	The smart home (Chapters 4 and 5)
NGINX Proxy Manager	Virtual machine	One front door for every web UI (6.3)
Pi-hole	Container	DNS for the whole house (6.1)
Step-CA	Container	The private certificate authority (6.4)

What a hypervisor buys you

A hypervisor runs other operating systems as *guests*. The alternative, one service per physical machine, costs a power supply, a network port and a backup plan per service. With Proxmox VE, one small box runs all four, each isolated from the others, each backed up the same way, and each restorable in a couple of clicks.

Proxmox gives you two kinds of guest:

- **Virtual machines (VMs)** emulate a whole computer with its own kernel. Use one when the guest wants to own its operating system. Home Assistant OS ships as a full appliance image and expects that, and so does the NGINX Proxy Manager appliance.
- **Linux containers (LXC)** share the host's kernel and are far lighter: a container might use 512 MB of RAM where a VM would want two or three gigabytes. Use one for a single Linux service that doesn't care about its kernel: Pi-hole, the certificate authority.

The rule I ended up with: **container by default; VM when the software ships as its own OS, needs its own kernel, or you want a hard security boundary.** Make containers *unprivileged* (the default), so root inside the container isn't root on the host. You'll meet the one cost of that in 6.4.

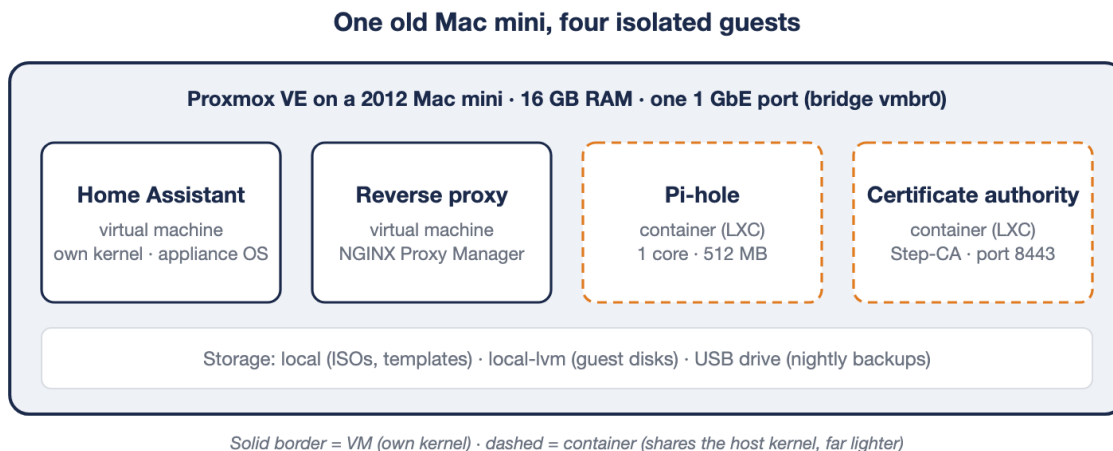


Figure 1.2 · *The first Proxmox host and its four guests.*

Installing on a Mac

Get it: Proxmox VE, proxmox.com · mbpfan, github.com/linux-on-mac/mbpfan · **Playbook:** handmedownhomelab.com/p/1.2-proxmox-mac-mini

Proxmox installs on a Mac mini like any PC: write the installer ISO to a USB stick, hold **Option** at power-on to pick the boot device, and follow the installer. Three things are Mac-specific or easy to miss:

1. **One network port for everything.** The 2012 mini has a single 1 Gb Ethernet port. Proxmox creates a bridge (`vmbr0`) on top of it, and every guest shares that uplink. For DNS, a proxy and a CA that's plenty. For any link-level check, such as reading which switch port you're on with LLDP, use the physical interface (`nic0`), not the bridge.
2. **The Wi-Fi radio is useless here.** Leave it alone.
3. **The fan.** As 1.1 said, the SMC's automatic curve gets no input under Linux. On this machine the fan sat at its 1,800 RPM minimum while a one-minute CPU load took the processor from 64 °C to 85 °C and still climbing. Nothing was in danger yet, but there was no cooling response at all.

The fix is `mbpfan`, a small daemon that reads the temperature sensors and drives the fan. It's an ordinary Debian package. Then set the curve in `/etc/mbpfan.conf`. I used 60/70/90 °C here:

```
low_temp = 60    # at or below: minimum fan speed
high_temp = 70   # above: start ramping
max_temp = 90    # at or above: full speed
```

That's higher than you'd pick for a desktop, on purpose: this box idles at 56–68 °C, and a lower threshold would have the fan audible around the clock. Under the same test load afterwards, the fan stepped from 1,800 to 2,412 RPM within ten seconds, and the peak was 80 °C instead of 85.

One more setting matters. Once `mbpfan` takes over, the fan is in manual mode, and if the daemon dies uncleanly the fan stays wherever it was, so give its service a systemd restart policy (`Restart=always`, in a small drop-in file).

The companion files do all of this in one step: `1.2/install-mbpfan.sh` installs `mbpfan`, reads your fan's own minimum and maximum speeds, writes the curve and the drop-in, and then checks the fan is actually being driven.

Your first guests

A container from a template, in the web UI: **local** → **CT Templates** → **Templates**, pick a Debian image, then **Create CT**. Give it a hostname, a root password or SSH key, one core, 512 MB of RAM, a few gigabytes of disk, and DHCP. A VM is the same flow with **Create VM** and an ISO or an appliance image. From the host's shell, `pct list` shows the containers and `qm list` the virtual machines:

```
pct list
qm list
```

Set **Start at boot** on every guest you care about. My certificate authority was the one guest without it, and after a routine reboot it simply stayed down until someone noticed (`pct set 200 --onboot 1` fixed it).

A fresh install gives you two storage pools: `local` for ISOs, templates and backups, and `local-lvm`, a thin-provisioned pool for guest disks. Backups get a third pool, on a separate disk. That's a subject of its own, and it's section 1.4.

One machine, three names

This host has three names, and tools will trip over the difference:

- **The Proxmox node name** is `myproxmox`, from the installer. API paths use it: `pvesh get /nodes/myproxmox/...` works, `/nodes/proxmox/...` fails.
- **In DNS, `proxmox.home.arpa` is the reverse proxy**, not this machine, because it serves the web UI with a real certificate (6.3). `ssh-proxmox.home.arpa` is the machine itself.
- **In my notes it's just "proxmox"**.

So anything that SSHes to this host uses `ssh-proxmox.home.arpa` or its IP address, never `proxmox.home.arpa`. The nightly backup copy in 1.4 uses the IP for exactly this reason: it once failed because the proxy name was pointing somewhere unexpected.

On a fresh build, avoid the first mismatch: give the installer the name you'll actually use. Renaming a Proxmox node later means moving its configuration by hand.

1.3 A second Proxmox host

One server is a single point of failure. So I added a cheap N100 mini PC, and then spent more time on its backups than on anything I run on it.

Siblings, not a cluster

Proxmox can join hosts into a *cluster*: one management view, live migration of guests between machines, shared configuration. Clustering needs at least three votes to stay safe (quorum), and it ties the hosts' fates together in ways that are hard to undo. With two small machines, the simpler design

is two **independent** hosts: separate installs, separate storage, separate backups, and no shared anything. If you ever want `qm migrate` between them, that's the moment to cluster them, on purpose.

What makes the second host useful is *what* goes on it: the services that should survive the first host going down. Here that's the second Pi-hole (6.2), the uptime monitor that watches everything else (Chapter 3), the central log collector, a sandboxed AI agent (Chapter 8), and Nextcloud (Chapter 9).

The box itself: an Intel N100 with four efficiency cores, 16 GB of RAM and a 1 Gb Ethernet port. Some N100 mini PCs ship with 2.5 GbE; this one doesn't, so check before you buy if that matters to you. It came with Windows on its SSD, which was wiped and reclaimed.

A storage layout that puts speed where it counts

Disk	Type	Holds
Internal laptop drive	5,400 RPM HDD	The Proxmox system itself
Internal M.2	SATA SSD	Every guest disk (a thin pool called <code>local-ssd</code>)
USB drive	5,400 RPM HDD	Backups

The host's own operating system does very little I/O, so it can live on a slow disk. The guests do all the work, so they get the SSD. Moving the system disk too would have meant a reinstall, and it wasn't worth it.

Drives by identity, never by letter

On one reboot, the two USB drives swapped letters: the one that had been `sdc` came up as `sdd`, and the other way round. Nothing broke, because every mount is by UUID. Letters are *discovery order*, not identity. To find a drive, use its model and serial number:

```
lsblk -o NAME,MODEL,SERIAL,SIZE,ROTA
```

One enclosure is one failure domain

For a while there were two drives in a cheap dual-bay USB enclosure. It looks like two independent disks. It isn't. The enclosure presents both drives through one USB bridge, on one cable, from one power supply, and when it re-enumerates, *every* drive in it drops at once. It also serializes them: the kernel reports `can_queue = 1` for the whole enclosure, so the two drives take strict turns rather than sharing bandwidth. On this host's workload (a three-minute nightly backup) the slowdown barely matters. The shared failure does. A second disk in the same enclosure protects you against one disk dying and nothing else. A real second copy lives on a different machine.

A trap if you use Proxmox's HA manager

One guest on this host, the AI agent VM, is registered with Proxmox's high-availability manager. On a single host that does one surprising thing: `qm shutdown 100` asks the HA manager to stop the VM, and that desired state is *persistent*. After the next reboot the VM stays down, even with `onboot` set, while every other guest comes back. Start it the HA way, `ha-manager set vm:100 --state started`, and check it with `ha-manager status`.

1.4 Backups that tell you if they fail

Both of my Proxmox hosts reported a successful backup every night for weeks while writing their backups to the wrong disk. Here is the backup design that came out of it, and the one setting that would have caught it on day one.

Get it: Proxmox VE's built-in backup, pve.proxmox.com/wiki/Backup_and_Restore · **Playbook:** handmedownhomelab.com/p/1.4-backups

The nightly job

Proxmox's built-in backup tool is `vzdump`. Schedule it under **Datacenter** → **Backup**. The backup storage is a plain directory on a USB drive, here mounted at `/mnt/usb-backup`. On the first host the job runs every night at 02:00, backs up all four guests, and takes about seven and a half minutes. It writes roughly 25 GB of compressed archives, and 18 GB of that is Home Assistant. Restoring is **Storage** → **usb-backup** → **Content**, pick an archive, **Restore**. Do that once, on purpose, before you need to.

Four settings are easy to get wrong:

- **Stagger the schedules.** The second host backs up at 02:30, the first at 02:00, so the two Pi-holes are never being snapshotted at the same moment.
- **Set compression.** The web interface's backup form defaults to `zstd`. A job created from the command line without it writes uncompressed archives, about three times the size (547 MB against 159 MB for the playbook test's small container).
- **Check the guest list.** Both jobs name their guests explicitly (`--vmid 100,120,101,200` on the first host, `--vmid 100,101,102,103,130` on the second), so a new guest is **not** backed up until you add its ID under **Datacenter** → **Backup** → **Edit**. Nothing warns you. The job keeps succeeding without it.
- **Bind mounts aren't backed up.** If a container mounts a folder from the host (Nextcloud's data drive, in Chapter 9), `vzdump` backs up the container and skips that folder completely. Those files need their own backup.

The one flag that matters: `is_mountpoint`

A USB drive plugged into a Linux box is mounted at a directory, and that directory exists whether or not the drive is there. If the drive drops off the bus, the kernel unmounts it, and the directory is left behind as an ordinary empty folder on the root disk. A backup job pointed at that folder writes into it happily, fills the root disk, and reports success, because writing to a folder *is* a success.

Proxmox has a guard for exactly this. Set `is_mountpoint 1` on the backup storage, and Proxmox marks the storage offline whenever the path isn't a real mount. The backup fails loudly and emails you, instead of landing on the root disk. **Set it on every removable backup storage, on every host, the day you create it.** The war story below is what happens when you set it on one host and not the other.

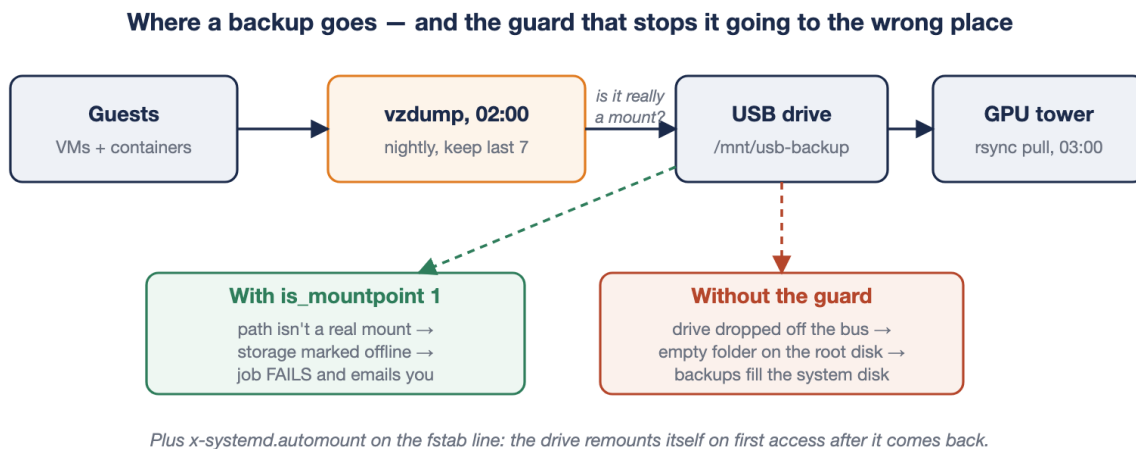


Figure 1.3 · The nightly backup chain, and what `is_mountpoint` protects against.

A backup mount that heals itself

The guard turns a silent failure into a loud one. The next step is to stop the failure happening at all. Add `x-systemd.automount` to the drive’s `fstab` line (mounted by `UUID`, with `nofail` and short device and mount timeouts) and `systemd` keeps a placeholder on the path permanently; the first time anything touches it, the real filesystem is mounted on demand. Proxmox checks its storage every few seconds, so in practice the drive is remounted within moments of reappearing, with no cron job and no human. Both hosts’ backup drives are mounted this way now. The first host only got it in October: its September fix added the guard and the driver quirk and missed this line, which is the war story’s lesson learned a second time.

Don’t take that on faith; test it. I tested three cases on the real host, and the playbook test repeated all three on a fresh Proxmox with the same results:

Scenario	Result
Normal operation	Storage active, all backups listed
Filesystem unmounted, drive still attached	<code>pvesm status</code> alone remounted it
Drive genuinely gone (USB unbound)	Storage marked inactive ; a write attempt fails with “No such device”; 0 KiB landed on the root disk. Rebinding the drive brought everything back intact

There was a worry that the permanent automount placeholder would make `is_mountpoint` always pass and defeat the guard. It doesn’t: with the device absent, the automount fails, and Proxmox marks the storage inactive. The two mechanisms work together. One job is still lost if the drive drops *during* a backup, and that job’s failure email is the signal.

One more defense: both enclosures’ USB bridges were moved off the `uas` driver with a kernel quirk (`usb-storage.quirks=152d:1561:u`, with each enclosure’s own IDs), which is a well-known fix for this disconnect pattern.

A copy on another machine

A backup on a USB drive plugged into the same machine is a rollback, not a backup. A second job on the GPU tower pulls the archives across the network an hour later. It's a *mirror*: `rsync --delete` makes the copy match exactly what Proxmox keeps, so the retention setting (keep the last seven) governs both copies. That's also its danger. A mirror copies a missing source as faithfully as a real one, so the script refuses unless the backup drive on the Proxmox host is really mounted and still holds at least half of what the copy holds. The whole script is three short guards and an `rsync`. It's `1.4/proxmox-backup-sync.sh` in the companion files, and the playbook installs it and tests every guard.

Cron runs it from a one-line entry. Leave off the usual `>> logfile` redirect: the script is silent when all is well, so anything it prints is a problem, and cron mails it to root (3.3 makes sure that mail arrives). Two details the playbook test turned up: the script counts only compressed (`.zst`) archives, so a backup job set to no compression makes it refuse every night; and with the self-healing automount, `mountpoint` says yes even when the drive is gone, and it's the `ls` beside it that catches the absence.

Note the IP address instead of a hostname in the script: 1.2 explained why.

WAR STORY · THE SAME FAILURE, TWICE

Symptom, first time. In late July, the newest backup on the second host's USB drive was 13 days old, yet every nightly job had reported success.

Wrong guess. The drive must be failing. SMART said otherwise: health passed, zero reallocated sectors, zero pending, zero errors.

Cause. The enclosure had dropped off the USB bus mid-write on 14 July and come back 15 seconds later. The kernel re-attached the disk every time, but nothing remounted the filesystem, so for 13 days the backups went into the empty directory on the 94 GB root disk: about 18 GB of them. Then a reboot remounted the drive *on top of* those files, hiding them. The drive showed stale backups, and the current ones sat invisible underneath, still using space. The tell, if you know to look: `pvesm status` showed the backup storage with exactly the same size and usage as `local`, because it *was* local. To see files hidden under a mount: `mount --bind /mnt/x`, then look in `/mnt/x/mnt/usb-backups`, that host's backup path.

Fix. `is_mountpoint 1` on the storage, the self-healing automount, and the USB driver quirk. Fixed on 27 July, on that host.

Symptom, second time. Six weeks later, one night in September at 01:35, the *first* host's USB drive dropped off the bus and reappeared a second later. Twenty-five minutes after that, the 02:00 backup job ran on schedule and reported success. The next morning the root disk was 70% full instead of 45%. The only alert came from `smartd`, which emailed that it couldn't read SMART data from the USB drive. That reads like a dying disk. The disk was fine.

Cause. Identical. Even the script written to apply the fix was fooled once: it logged "mount returned 0" while nothing was mounted, because the drive hadn't finished reappearing. The `fstab` line carries `no-fail`, and with `nofail`, `mount` returns success when the device is absent. That's what the option means.

And the copy on the tower? At 03:00 that night the mirror did exactly what it was told. It copied the one night of archives sitting on the root disk and deleted every night the tower already had. The next night, with the drive remounted, it copied the week back. Nobody noticed for almost three weeks, until a review of this chapter turned it up. Had the drive died instead of dropping, the "off-box backup" would have held a single night. That's why the script above has a guard. It didn't then.

Lesson. Two of them. An exit status answers the question the command was asked, not the one you meant; and if this ever happens to you, clean up in the right order: move the misplaced archives *off* the root disk before remounting, because mounting over them hides them while they keep using the space. And the one that stings: the fix was applied on 27 July to the host that bit me, and never to the other. When you fix a class of problem, sweep every machine with the same shape, not just the one that failed.

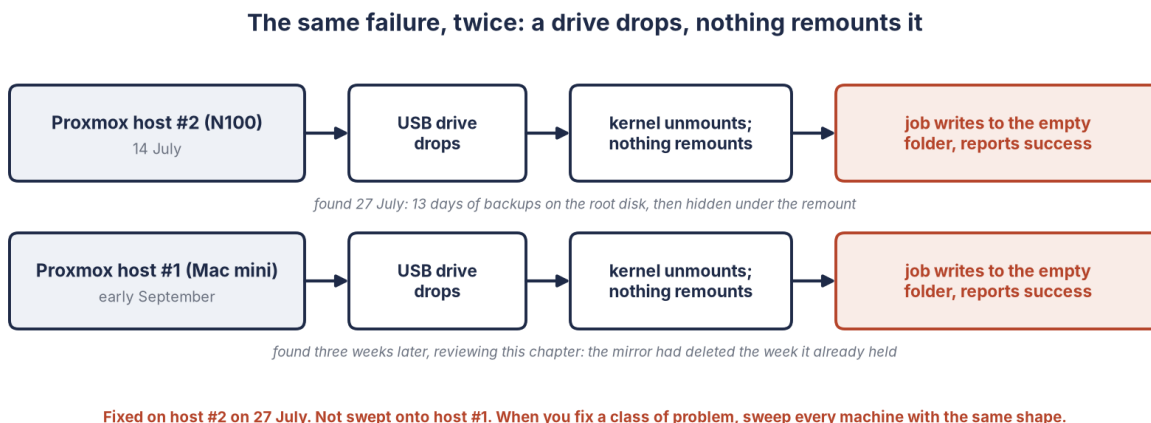


Figure 1.4 · The same failure, on both hosts, six weeks apart.

1.5 The gateway, and DHCP reservations as code

The house has 104 devices with fixed addresses. Nobody clicked 104 times. There's a table in my notes, and a script loads it onto the router.

Why the consumer router had to go

The house ran on an Eero mesh, and it's a good product for what it's for. A homelab asks for things it won't do:

- **No way to export the reservation table.** Every fixed address lived only in a phone app.
- **A 1 Gb WAN port**, capping the internet connection below what the plan delivered.
- **No port mirroring**, so no way to feed a copy of the traffic to an intrusion-detection sensor (Chapter 11).
- **Cloud-dependent configuration.** Every change goes through the vendor's cloud, which matters during a cutover, as you'll see.

Choosing the replacement

The research first pointed at a UniFi Cloud Gateway Max. The purchase was a UniFi Express 7 (UX7) at about \$199 instead. Both have the same routing engine and the same intrusion-detection throughput, so for this house they route identically. The Express 7 adds a Wi-Fi 7 radio, and because the new gateway would sit centrally in the basement, next to where the internet comes in, that radio could replace a badly placed mesh node.

Its one weakness is a single LAN port. A small managed switch fixes that: a UniFi Flex Mini 2.5G takes the gateway's trunk on port 1, fans out to the rest of the house, and **mirrors port 1 to port 3**,

where the intrusion sensor listens. That mirror is what made the switch worth buying. It's the first of several small managed switches: the others hang off it, in the equipment rack and around the house, which is why later chapters mention more than one.

One expectation to set before the first speed test: every host in this house has a 1 Gb port, so **no single machine will ever see more than about 940 Mb/s**. The faster WAN is an *aggregate* gain, several devices at once. A laptop reading 940 after the swap is correct, not a failure.

The addressing plan: why a /22

The house network is one flat /22: 10.20.0.0 to 10.20.3.255, a single network of about a thousand addresses. It's one broadcast domain, not four subnets; the third number carries no meaning. A /22 gives room for 100+ reservations *and* a large pool for everything else without ever running out. Two settings UniFi won't guess for you:

- **The gateway keeps the old router's address**, 10.20.0.1. Every device with a hand-set address points at it. Keeping it is what let all of them survive the swap untouched.
- **Define the pool explicitly**. UniFi won't auto-size a /22. Here the pool is 10.20.0.16–10.20.3.254, with a 4-hour lease.

Cutover day

The work was split so that the outage itself was short: about 15–30 minutes in the daytime.

Before the day (zero downtime). Configure the new gateway on the bench, disconnected from the house. Clone the old router's WAN MAC address so the internet provider hands out the same public IP. Set the LAN, the pool and the DNS. For the bench, use a *public* DNS resolver: the Pi-holes are on the old network and unreachable from the bench, and pointing DHCP at them there breaks DNS in a thoroughly confusing way. They go in last. Adopt the switch and set up the mirror while you're at it. And **load every reservation before the gateway touches the house network**, so each device lands on its correct address on its very first DHCP request.

The two ordering traps.

1. **Put the mesh into bridge mode *before* you move the modem cable.** The Eero app needs the cloud to push that change. Move the modem first and the Eeros are offline and can't be changed at all.
2. **Never let two routers hand out addresses on one network.** Two DHCP servers is chaos. Bridging the old mesh is exactly what prevents it.

Trap 1 fired on the day. The modem moved first. Recovery didn't need a factory reset: modem back to the Eero, bridge it in the app, modem back to the new gateway. It cost one round trip. The detail worth remembering: the Eeros *looked* fine in the app on a phone over cellular. Seeing them isn't the same as being able to change them.

After. Spot-check a handful of hosts on their documented addresses. Switch DHCP's DNS to the Pi-holes as the very last step. Then write down your public IP address. I didn't record it beforehand, so the "same public IP after the swap" check could never be verified.

Reservations as code

The reservation table is a Markdown table in the wiki, one row per device:

Device	IP	MAC
pihole	10.20.3.173	bc:24:11:...
ha	10.20.3.172	bc:24:11:...

A Python script, `ux7_load_reservations.py`, reads that table and makes the gateway match it through the UniFi API. The design choices are the useful part, and they transfer to any gateway with an API:

- **Dry run by default.** Nothing is written without `--apply`. A dry run lists every row as CREATE, UPDATE or SKIP.
- **Idempotent.** Run it twice and the second run does nothing. Re-run it after every edit to the table.
- **No passwords on the command line.** The script reads the gateway password from an environment variable, then the macOS Keychain, then a prompt, in that order. It needs a *local* UniFi admin account; a cloud account with two-factor authentication can't log into the local API.
- `--list` parses the table offline and prints what it found: a check that the table itself is sane before touching the network.
- `--verify` is read-only and checks the gateway's live settings (subnet, gateway address, lease, pool, DNS, reservation count) against what they should be. Run it after every firmware update; an upgrade can silently reset a setting.
- `--prune` finds reservations on the gateway that are no longer in the table.
- **Each row is routed by its own address.** When the IoT network arrived (1.6), the script learned to send a row in 10.20.50.x to that network and a row anywhere in 10.20.0.0–10.20.3.255 to the main one. A row for a network the gateway doesn't have stops the whole run before anything is written.

The typical session: parse the table offline, dry-run against the gateway, write, then check read-only:

```
./ux7_load_reservations.py --list
./ux7_load_reservations.py
./ux7_load_reservations.py --apply
./ux7_load_reservations.py --verify
```

The script is `1.5/ux7_load_reservations.py` in the companion files, with an example table to start from. Edit the block marked *YOUR NETWORK* first.

AI AT THE KEYBOARD · HOW THE LOADER WAS WRITTEN

I didn't write `ux7_load_reservations.py` by hand. The AI wrote it the day before the cutover, to replace a hundred reservations' worth of clicking: read the table in the wiki, make the gateway match it, dry run by default. It checked the parser against the live table (104 rows, 3 retired devices skipped, no duplicate addresses) before anything touched the gateway. My part was the decisions, such as whether to reserve addresses for cloud-only gadgets that gain nothing from one (I said yes, since it cost nothing once scripted), and reading every dry run before saying yes.

One thing it can't do: UniFi's own switches and access points refuse reservations entirely. The controller owns their addresses, and the gateway rejects the attempt with `FixedIpAlreadyUsedByDevice`. Leave them out of the table.

WAR STORY · "NOTHING WAS WRITTEN"

Symptom. I replaced my iPhone and needed its reservation moved to the new phone's MAC address. The load failed with `DuplicateFixedIP`, naming the *old* phone.

Wrong guess. The loader is broken.

Cause. The gateway refuses a second reservation on an address that's already reserved, so the old one had to be cleared first. I ran `--prune`, then `--apply`. On a dry-run-by-default tool, that's a prune *dry run* followed by a load. The prune's report ended "Nothing was written. Re-run with `--apply` to commit", at the bottom of a report that otherwise read like a finished job.

Fix. `--prune --apply`, then `--apply`. The proof was on the tool's read path, not its write summary: the phone's row flipped to SKIP.

Lesson. A safe default only stays safe if every step respects it. Each subcommand needs its own `--apply`, and the failure shows up one step *after* the mistake, in a different command.

1.6 An IoT VLAN, and the camera VLAN I canceled

Every tutorial says to put your cameras on their own VLAN. I designed it, planned it, and canceled it. Here's what I did instead, and why.

The flat-network problem

On a flat network, every device can talk to every other. A cheap smart plug with abandoned firmware can reach the hypervisor's web interface, the DNS server, and your laptop. A **VLAN** (virtual LAN) splits one physical network into several logical ones. Traffic between them has to go through the gateway, and the gateway's firewall decides what's allowed.

That last sentence hides the rule that makes the whole thing work: **enforcement needs a routed boundary**. Two groups of devices on the same network share a broadcast domain, and their traffic is switched directly between them. It never reaches the firewall, so no rule can touch it. A separate network is the only way to put a firewall between two groups.

How a device gets its VLAN

A wired device gets its VLAN from the switch port it's plugged into. A **Wi-Fi device gets its VLAN from the SSID it joins**. Each wireless network name is bound to one VLAN. For a house full of smart-home gadgets that are nearly all Wi-Fi, that changes everything about how you migrate them.

The plan

The original design split the house into:

Network	Who	Rule
Infrastructure + trusted	Hypervisors, servers, laptops, phones	Reaches everything
Cameras	Cameras and the video recorder	No internet
Media	TVs, streaming boxes	Internet, plus Pi-hole DNS
IoT	Plugs, bulbs, appliances, sensors	Internet, plus Pi-hole DNS
Guest	Visitors	Internet only

Why the camera VLAN died

When it came time to build the camera network, three things came out of the live controller that the plan hadn't known:

1. **Most of the cameras weren't wired.** The plan assumed wired cameras. Of the twelve devices assigned to it, seven were Wi-Fi. A camera VLAN meant a third SSID and re-on-boarding every one of those cameras by hand.
2. **The security goal was already met.** The point was “fixed cameras get no internet access.” An address group listing the eight fixed cameras, plus one firewall policy blocking that group from the internet, already did exactly that.
3. **It would have cost real traffic.** The cameras send about 591 GB a day to the video recorder, and the AI detector pulls another 104 GB, all switched locally today. Splitting cameras from the recorder makes the gateway route all of that.

So it was canceled, not postponed, and one rule survives it: **cameras and their recorder stay on the same network.** Chapter 10 builds on that rule.

The move that worked: rebind the SSID, not the devices

An earlier plan had died the same way. It would have moved about 70 devices to a new SSID, one at a time, re-entering the Wi-Fi password on each. The version that shipped turned it inside out. Instead of moving the devices to a new network name, it moved the *network name*: the house's existing SSID was rebound to the new IoT VLAN, which relocated about 40 clients by changing one setting. Then a brand-new SSID on the main network was created for the handful of devices that genuinely need full access, and those few were re-joined by hand.

Two conditions make that trick safe. The IoT side must be the side that rennumbers, because putting the *trusted* devices on the new VLAN would leave the gadgets sharing a network with the servers, which defeats the point. And anything that needs a stable address on the new network needs its reservation rows written *before* the cutover, which is why 1.5's script learned to route rows by address.

The firewall rules that go with it:

- **IoT → internet:** allowed.

- **IoT → Pi-hole DNS:** allowed (and ping, for the health checks).
- **IoT → Home Assistant's MQTT broker:** allowed. This is how ESPHome and WLED devices report in (Chapter 4).
- **IoT → everything else on the main network:** blocked.
- **Main network → IoT:** allowed, so Home Assistant can control the devices.

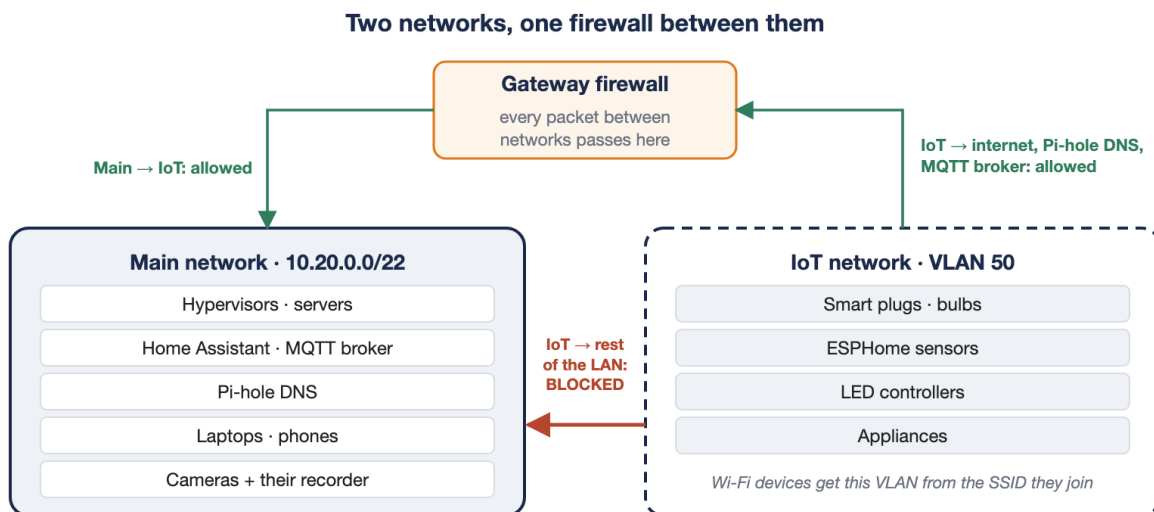


Figure 1.5 · The main network and the IoT VLAN, with the firewall rules between them.

The morning after, the reservation loader dry-ran to zero changes, 33 devices were on the new network, and Home Assistant showed all 108 of its TP-Link entities and all 15 ESPHome entities available. Then the containment check, from a real device *on* the IoT network with a shell (the radio Pi from Chapter 9):

Target	Expected	Got
Proxmox web UI	Blocked	Blocked
GPU tower SSH	Blocked	Blocked
Home Assistant web UI	Blocked	Blocked
Gateway admin	Blocked	Blocked
Pi-hole DNS, MQTT, internet	Allowed	Open

Moving a wired device: plan a cable pull

Changing a switch port's VLAN in UniFi doesn't drop the link, so a wired device on that port keeps its old address and becomes unreachable from *both* networks. A remote port bounce didn't help. Pulling the cable for ten seconds did. Before you move a wired device, arrange for someone to be standing next to it.

When a design isn't worth its cost

The camera VLAN is the useful failure here. A design can be correct and still not be worth building, if the thing it protects is already protected another way and the migration costs more than the risk it removes. Write down why you declined it, so nobody reopens it next year.

WAR STORY · THE FIREWALL THAT WASN'T BLOCKING ANYTHING

Symptom. On cutover night, a port check from the laptop said every TCP port on the IoT network was closed: cameras, ESPHome devices, even the gateway's own DNS. Ping worked.

Wrong guess. A missing or misordered firewall rule. "ICMP up, all TCP down" looks exactly like that, and it was already being written up as a regression.

Cause. The laptop was on Wi-Fi. The same checks from two wired servers came back open. The rule had been fine all along; the Wi-Fi path from the main network to the IoT network gave a uniform false negative. Why was never fully explained. The pattern held across five targets and three source machines.

Fix. Test from a wired host, and test containment from a device that's actually *on* the restricted network.

Lesson. A probe tells you about its own path before it tells you about the thing you're probing. Choose the vantage point before you trust the answer. (The same trap bites again in Chapter 10.)

Before you move on

- ☐ Every machine sized by its binding constraint, with a wiki page.
- ☐ Proxmox with guests set to start at boot.
- ☐ Backups on a separate disk, guarded with `is_mountpoint` on **every** host, self-healing, copied to a second machine, and one restore tested.
- ☐ Address reservations kept in a file and loaded by a dry-run-first script.
- ☐ Untrusted devices on their own VLAN, and containment tested from a device on it.