PRIVATE AI FOR SMALL BUSINESS

Private AI for Small Business

Run your own AI on hardware you own — so nothing leaves the building.

RobotNurse

Free PDF edition · robotnurse.io

A note on what this book is, and what it isn't.

This book is for people who run or own a small professional practice — a medical or dental office, a law firm, an accounting firm, a boutique clinic, a solo practice with a few employees. You are not a hobbyist. You don't care about tokens-per-second or kernel fusion. You care about one question: *can I get the productivity of a modern AI assistant without handing my clients' data to someone else's server?*

The answer is yes — with caveats this book will state plainly. Everything in these pages comes from real, hands-on work: roughly a year of running open-weight language models on our own hardware — two GPU machines, two Apple Silicon workstations, and the software around them. We kept a field notebook the whole time. Most of the entries are about things that went wrong, or things we measured and were surprised by. That is exactly the material a decision-maker needs, because the marketing for "private AI" is full of claims nobody tested.

This book contains no invented benchmarks, no fabricated statistics, no customer testimonials, and no hype. Where a number appears, it is either something we measured ourselves or a price you can verify with a quick search. Where we don't know something, we say so.

The pen name on the cover is **RobotNurse**. The book is written by an engineer who installs and maintains local AI systems for small practices. The advice here is the same

> advice given on-site: blunt, specific, and honest about what breaks.

Contents

1. The Cloud Is a Leak
2. What "Private AI" Actually Means
3. The Hardware That Runs What
4. The Setup, End to End
5. Privacy and Offline Operation
6. Trusting What the Machine Says
7. What It Actually Costs
8. Keeping It Running
9. The Decision Checklist

Appendix: Where the numbers come from

How to read this book

The book is organized so you can read it two ways. Read it front to back if you're deciding whether this is even for you — the argument builds. Or, if you're already convinced and just need to act, jump straight to Chapter 9 (the decision checklist), then Chapter 3 (what to buy) and Chapter 4 (how to set it up), and treat Chapters 6 and 8 as the reasons those other chapters say the careful things they say.

Every chapter is self-contained enough to read on its own. Technical terms are kept to a minimum and explained where they appear; the glossary at the back covers the rest. Where a chapter

makes a factual claim, the appendix "Where the numbers come from" points back to the underlying field notes.

CHAPTER 1

Chapter 1 – The Cloud Is a Leak

The quiet problem with the assistant everyone uses

Here is the ordinary way a small professional firm adopts AI today. Someone reads about a new assistant, opens an account, pastes a contract into it to "summarize this," pastes a client's medical history into it to "draft a referral letter," pastes a tax return into it to "explain this to the client." It works. It is genuinely useful. The firm is now more productive. And every one of those documents has been sent to a server the firm does not own, does not control, and has not vetted.

That is the whole problem in one paragraph. It is not that cloud AI is unreliable – it is often very good. It is not that cloud AI is insecure in a dramatic, hacker-movie sense. It is that **using cloud AI means your clients' data is no longer inside the building, and for a regulated practice, "inside the building" is not a preference – it is the job.**

What "regulated" actually requires

Three professions illustrate the point clearly, because each one has a written, enforceable obligation to protect client information.

Medicine. In the United States, health-care practices are subject to HIPAA. A covered practice that sends protected health information to a third-party service needs a Business Associate Agreement (BAA) with that service. A BAA is not a checkbox; it is

a legal contract that makes the vendor accountable for the data. Many consumer AI assistants either will not sign a BAA for a small practice, or will sign one only on an enterprise tier that a three-person clinic cannot justify. The uncomfortable fact is that a lot of medical AI use happens without a BAA at all — the doctor pastes the patient note into a consumer assistant and trusts that "the vendor says it's private." That is not compliance. That is a disclosure waiting to happen.

Law. Attorneys owe clients a duty of confidentiality and, in many places, a duty of technological competence. Sending privileged client communications to a third-party AI service without a confidentiality arrangement can waive nothing by itself, but it exposes the firm to a serious argument that reasonable care was not taken. The bar rules are still catching up to the technology, which is precisely why the safe default is to avoid the question entirely: keep the data on hardware the firm owns.

Accounting. Tax professionals hold some of the most sensitive data a person has — identity documents, income, Social Security numbers. IRS rules (including the "safeguards rule" and the requirements around taxpayer data) and professional standards like those from the AICPA require the practitioner to protect it. The same story applies: sending returns and source documents to a third-party model is a real exposure, and the practice carries the liability, not the vendor.

Notice what all three have in common. The *accuracy* of the AI is a secondary concern next to the *custody* of the data. You can argue

about whether the model occasionally hallucinates. You cannot argue about where the bytes went.

The liability is yours, not the vendor's

There is a structural reason this problem keeps getting deferred, and it's worth naming. When a cloud vendor mishandles data, the practice that sent the data is the one that faces the client, the regulator, and the disciplinary board. The vendor's terms of service almost universally disclaim responsibility for your compliance with *your* obligations. Read any of them: the "you are responsible for how you use this service" language is standard, and it means exactly what it sounds like.

For a medical practice, that means the physician — not the vendor — answers for a patient-record disclosure. For a law firm, it means the firm — not the vendor — answers for a confidentiality breach. For an accountant, it means the preparer answers for the taxpayer data. The vendor can be a competent, well-intentioned company and this remains true, because the obligation attaches to *you* by law, and to them only by contract.

This is the crux of why "private" matters more than "powerful." You cannot outsource the legal obligation. You can only change the facts on the ground — and the only fact that actually helps is keeping the data on hardware you control.

A concrete picture

Imagine a two-person dental practice. The office manager pastes a patient's treatment-plan note into a consumer AI assistant to draft

a letter to the referring physician. It works beautifully. Three months later the assistant's vendor updates its data policy, and the practice discovers the notes were used to improve the vendor's model, or were accessible to a subcontractor, or were subject to a broadened retention clause. None of this is unusual; all of it has happened in various forms across the industry.

The practice's exposure is now real, and it has nothing to do with whether the model was accurate. It has to do with where the bytes went, and whether anyone can prove they were handled in a way the practice's obligations permit. That proof is what the practice owes — and it is a proof the practice simply cannot produce, because it never controlled the data in the first place.

The two objections, and why they don't hold

When this is raised, two defenses come up. They deserve an honest answer.

"The vendor says it doesn't train on my data." Good — and you should prefer vendors who make that commitment in writing. But "we don't train on it" is not the same as "the data never leaves your building." The data still transits to a server you don't control, is processed in memory on that server, is stored there (if only transiently), and is subject to that vendor's security, that vendor's subpoenas, that vendor's employees, and that vendor's terms-of-service changes. "We don't train on it" answers one narrow question. It does not put the data back under your roof.

"I'll anonymize it first." In practice, almost nobody anonymizes well. Real documents are dense with identifiers — names, dates, account numbers, addresses, the specific details that make a client *this* client. Stripping them by hand is slow and error-prone, and an automated "scrub" you didn't verify gives you the feeling of privacy without the fact of it. The honest version of this defense is: "if the data has been scrubbed so thoroughly that no one could identify the client, then it probably isn't the data I needed help with anymore."

The point of this book

This is not an argument that cloud AI is evil, or that you should never use it. It is an argument that **a regulated small practice has a specific, solvable problem that cloud AI does not solve**: the custody problem. The solution is to run a capable model on hardware you own, in your own office, with the network cut so the data physically cannot leave.

That is no longer a fantasy reserved for a data center. It is a machine that fits on a desk and costs less than a good laptop. The rest of this book is about what that machine actually is, what it can and cannot do, what it costs, and how to decide if it is right for your practice.

CHAPTER 2

Chapter 2 – What "Private AI" Actually Means

Four terms, separated clearly

The phrase "private AI" is used loosely, and the looseness is where people get misled. Four ideas get conflated. Let me pull them apart, because the whole decision rests on the difference.

Local means the model runs on your own hardware – a computer in your office – rather than on a remote server. It says where the computation happens. It does not, by itself, say anything about whether the machine is allowed to phone home.

Offline means the machine is not connected to the internet at all, or is connected only through a controlled, audited path. Offline is a stronger statement than local: a local model on an internet-connected computer is private-ish; a local model on an air-gapped computer is private in a way you can actually verify.

Open-weight means the model's parameters – the file of numbers that makes the model what it is – are published and downloadable. You download the file once, and from then on you own a copy. Nobody can change it under you, revoke access, or push an update that alters its behavior. This is different from "open source" in a licensing-lawyer sense, but for your purposes the practical fact is the same: the model is a file on your disk, and it stays the same file tomorrow.

Closed API means you don't download anything; you send requests to someone's server and receive answers. This is the cloud model again, just dressed in a nicer interface. It is the opposite of the first three.

A private-AI setup worth the name is **local + offline + open-weight**. The model file is on your machine, the machine can run it without the internet, and the file itself is a fixed artifact you control. When all three are true, you can state something most vendors cannot: *no data leaves the building, and we can prove it by watching the network.*

What you are actually buying

It is important to be clear-eyed about what this buys you, because the honest answer is narrower than the marketing suggests.

You are buying **custody and control**, not a better model. The best open-weight models today are very good — good enough for summarizing, drafting, coding, and question-answering in a professional context — but a leading closed model from a big lab is usually still a bit ahead on the hardest reasoning tasks. You are trading a small amount of raw capability for a large amount of control. For a regulated practice, that is almost always the right trade, but you should make it with your eyes open rather than believing the open model is "just as good at everything."

You are buying **predictability**. An open-weight model is a fixed file. If it gives you a correct answer on Tuesday, it gives you the same answer on Wednesday, because nothing changed. Cloud

models are updated under you; a prompt that worked last month can behave differently this month and you will not be told. For professional work where consistency matters — where you build a workflow around the tool — that fixedness is worth more than a percentage point of benchmark score.

You are buying **a clean answer to a compliance question**. "Where is our clients' data?" becomes a question with a one-word answer: here. That single fact simplifies an enormous amount of audit, policy, and anxiety.

The model is a file, and files are predictable

The most underrated property of open-weight models is that they are *files*. A 7-billion-parameter model, quantized, is about five gigabytes of numbers on your disk. It does not change unless you download a new copy. It has no terms of service. It cannot be revoked. It cannot be modified under you by someone pushing a silent update to a server.

We learned how much that matters the hard way, on the other side of the fence. During our evaluation work, we kept seeing bizarre, inconsistent results — models that should have behaved identically were behaving differently on different days. We traced it, finally, to a component in the *pipeline around the model*, not the model itself: a gateway was rewriting our temperature setting — the parameter that controls how random the model's output is — from 0 to 1.0, silently. Our "deterministic" setting was being ignored, and the model was sampling freely. We only caught it

because we were running the numbers ourselves and the numbers stopped making sense.

The lesson is not "temperature is important." The lesson is that when you rent a model through a vendor's pipeline, the vendor controls pieces you can't see, and those pieces can change without notice. When the model is a file on your own machine, there is no pipeline between you and it except the one you built, and you can inspect every part of it. For a practice that needs to trust its tools, that is the difference between "probably consistent" and "verifiably consistent."

The honest limits

Three limits deserve to be named now, so they don't surprise you later.

A local model is still a statistical system, not an oracle. It will occasionally be wrong, sometimes confidently. Chapter 6 is devoted to this, including the specific, measured ways we have seen these models mislead us. Running the model locally fixes the custody problem. It does *not* fix the reliability problem. You still need a human in the loop for anything that matters.

A local model knows only what you give it. It does not browse the internet by default, and it has no live knowledge of this morning's news. For many professional tasks — drafting, summarizing, reformatting, explaining a document — that is fine. For anything requiring current information, you will need a separate, deliberate step.

You become the IT department. This is the cost nobody puts in the brochure. When the machine breaks, or a model needs updating, or a new employee needs access, *someone* has to handle it. Chapter 8 covers what that actually looks like and how to keep it manageable.

CHAPTER 3

Chapter 3 — The Hardware That Runs What

The one number that matters: memory

Forget clock speeds, cores, and benchmark scores. For running a large language model locally, the number that decides almost everything is **how much memory the machine has, and how fast that memory is.** The model has to live in memory while it runs; if it doesn't fit, it doesn't run — or it runs so slowly it might as well not.

Here is the working formula, in plain terms.

A model is a file of numbers called parameters. A "7-billion-parameter" model is a file containing roughly 7 billion numbers. How much memory that file occupies depends on the precision at which you store those numbers:

- At **q4** (4-bit quantization, the standard "small file" setting): about **0.6 bytes per parameter.**
- At **q8** (8-bit): about **1.1 bytes per parameter.**
- At **fp16** (full 16-bit precision): **2 bytes per parameter.**

So a 7–8 billion-parameter model needs about **5 GB at q4, or 9 GB at q8**. But that's only the model file. While it runs, you also need memory for the "KV cache" — the model's working memory

of your conversation — and for the operating system and runtime overhead.

- **KV cache:** budget **1–4 GB** for a mid-size model with a reasonable context window.
- **Overhead:** budget **3–6 GB** for the operating system and the inference software.

Add it up. To run a 7–8B model comfortably at q4, you want roughly 5 GB (weights) plus a few gigabytes of headroom — a **16 GB** machine is the floor, and **32 GB** is comfortable. Here is the full picture, using the same arithmetic for the common model sizes:

- **7–8B parameters:** ~5 GB at q4, ~9 GB at q8
- **13–14B parameters:** ~9 GB at q4, ~15 GB at q8
- **30–34B parameters:** ~20 GB at q4, ~34 GB at q8
- **70B parameters:** ~40 GB at q4, ~75 GB at q8

The rule of thumb that falls out of this: **size the machine for the model one step above the one you think you need.** People reliably underestimate. You buy a machine that fits a 7B model today, and in six months you want to run a 13B model and the machine can't. Buy headroom.

The second number: memory *speed*

Memory capacity decides what fits. Memory bandwidth decides how fast it talks. This is where the two families of machines differ

in ways that matter for your use case.

- **Apple Silicon (M-series Macs)** uses "unified memory" shared between the CPU and the graphics processor, with bandwidth around **400–800 GB/s**. That is fast – fast enough that a 7B or 13B model feels genuinely responsive, and a 30B model is usable. Unified memory also means a Mac can allocate a *lot* of its memory to the model, which is why a Mac with 64 GB can run models a PC with 16 GB of video RAM cannot touch.

- **Dedicated GPUs (a PC with an Nvidia or AMD card)** have video RAM (VRAM) with bandwidth of **1 TB/s and up** – faster still, but limited in *amount*. A mid-range card holds 16 GB. That fits a 7B model comfortably, and a 13B model at a push, but not a 30B or 70B model, no matter how fast the card is.

- **Ordinary system RAM (DDR5)** is the fallback for machines without unified memory or a big GPU. It is an order of magnitude slower – around **100 GB/s** – which makes it fine for *loading* a model but painful for *talking to* it.

The practical takeaway: for a small professional practice, **a Mac with a lot of unified memory is the single best starting point**, because it runs a genuinely useful model, silently, in a small box, with a low power draw. A GPU PC is the choice when you need maximum speed or specific software that only runs on Nvidia. We'll get to specific machines and prices in a moment.

Three tiers, with real prices

Here is the concrete buying advice, organized by budget. Prices are street prices at the time of writing and will drift; the *relative* picture is what matters.

Tier 1 – under $800: the Mac mini M4 with 32 GB. This is the entry point that is actually worth buying. A 32 GB Mac mini runs a 7–8B model at q8 quality, or a 13B model at q4, and it does it silently while drawing around 30 watts. For a solo practice that wants to summarize documents, draft correspondence, and answer questions about a client's file – all offline – this is the machine. It is small, cheap, and quiet, and it holds its value.

Tier 2 – roughly $1,600–$1,900: a real step up. Two paths. Path A is a GPU PC built around a 16 GB card (an RTX 5070 Ti-class card, which has been around $900 on the street) plus 64 GB of system RAM. Path B is a Mac mini M4 Pro with 64 GB of unified memory. The GPU PC is faster on its card; the Mac fits larger models because 64 GB of unified memory beats 16 GB of VRAM when the model is big. For most practices the Mac path is simpler to own.

Tier 3 – around $3,000: the Mac Studio M4 Max with 64 GB. This is the "buy once, run the big model" option. It runs a 30B model with a long context window, or two models at once, and it does it at a power draw that is almost comically low for what it delivers. If you want to run the best open-weight models at comfortable quality and never think about the hardware again, this is the machine.

Memory bandwidth, in practice

The 400–800 GB/s figure for Apple Silicon is real, but the fine print matters. We measured a Mac's effective bandwidth warm versus cold and found a meaningful gap — roughly 850 GB/s with the system cache warm versus about 600 GB/s cold. The difference comes from where the data is sitting when you ask for it. In plain terms: the first question you ask a fresh model is slower than the ones after it. This isn't a defect; it's just how caches work, and it's the reason "feels responsive" is a warm-start property.

It also matters that **"fast enough" is a yes/no question, not a sliding scale.** A model either answers fast enough that you'll actually wait for it, or it doesn't. We spent time chasing throughput improvements — evaluating an alternative inference engine, and rejecting it because while it was 4–8% faster on one phase of work, it was 30–40% *slower* on the phase you feel in real time. The numbers can look good on paper and worse in the chair. This is why Chapter 4's "test it on your work" step is not optional.

One more subtlety worth knowing before you buy, because it can make an otherwise-fine machine look broken: **macOS throttles background work by design.** A process running "in the background" at a low priority (which the operating system does quietly to keep the machine responsive) can draw *zero milliwatts of GPU power* under load — we measured exactly that — because the system had deprioritized it. If you run your model at the default priority and it suddenly crawls for no apparent reason, the answer may be the operating system being polite, not a hardware failure. Set the inference process to a normal (or elevated) priority, and re-test.

What happens at the extreme

It is useful to know what the failure looks like when you push too far, because it teaches you where the real limits are. We tried to run a 192-billion-parameter model on a 256 GB machine — the kind of thing that, on paper, should just fit. It did not. We hit three separate blockers, and one of them was a hard stop that has nothing to do with how much memory you bought: macOS has a limit on "wired" memory (memory the system won't let be paged out), and our setup tripped it and **kernel-panicked the machine.**

The point of telling you this is not to send you looking up "wired memory." It's to make the Chapter 3 rule concrete: the number on the box is not the number you can actually use, and the largest models live on the far side of a line that desk hardware, however generous, is not built to cross. Buy for the model one step above your need, and stop there.

What these machines will *not* do

To keep expectations honest: none of these machines is going to run the very largest models (a 70B-class model wants ~40 GB at q4, which is right at the edge of even a 64 GB Mac Studio). None is going to serve dozens of simultaneous users. None is going to train a new model from scratch. Those are data-center jobs. What a desk machine *will* do is give you a private, offline assistant for one practice, or a small team, at a quality that — for drafting, summarizing, and document work — is genuinely good.

Chapter 4 – The Setup, End to End

The shape of the whole job

People imagine setting up local AI is an engineering project that takes weeks. It isn't. For a standard machine, the whole thing – from sealed box to a working, offline assistant – is an afternoon of work, with most of that afternoon spent waiting for a download. The steps below describe the real process, in order, including the parts that go wrong.

Step 1: Buy the machine (and don't cheap out on memory)

Using Chapter 3: pick the tier, and when you spec the machine, put every spare dollar into **memory**, not storage or CPU. Memory is the thing that determines which models you can run. Storage is cheap to add later; memory is not.

Step 2: Install the inference software

An "inference engine" is the program that loads the model file and runs it. The two you will most likely use are **Ollama** and **LM Studio**. Both are free, both are a single download, and both have a one-click install for Mac and Windows. Neither requires a terminal or any technical background to get running. You install the program, and it presents a small chat window where you can talk to a model immediately.

This is the step where people are consistently surprised that it is just... an app. No server rack, no Linux, no command line.

Step 3: Download a model

Inside the app you pick a model from a list and download it. A 7B model at q4 is a roughly 5 GB download; a 13B is around 9 GB. On a normal office connection this is minutes, not hours. You want to download the **q4** or **q8** version for your first model – the smallest file that runs well, because it is the easiest to verify working before you experiment with bigger files.

One decision to make early: which model. The names change faster than any book can keep up, so the durable advice is this: pick the most popular, most-downloaded general-purpose model in the size class that fits your machine. The popular ones are popular because they work well across the kinds of tasks a practice does – writing, summarizing, answering questions – not because they win any single benchmark.

Step 4: Test it against *your* work

This is the step that separates a useful tool from a toy, and it is the step most people skip. Do not test the model on "write me a haiku." Test it on the actual thing you will use it for. Paste in a real (or realistically disguised) client document and ask it to do the real task – summarize it, draft the letter, explain the clause.

Watch for two things. First, *speed*: is the answer coming back at a pace you would actually tolerate in a workday? Second, *quality on your task*: a model can be excellent at general writing and

mediocre at the specific thing your practice does, or vice versa. You will only find out by trying your actual work.

Step 5: Turn off the network (the part that matters for compliance)

This is the step that makes it *private* rather than merely *local*. Once the model is downloaded and working, you cut or restrict the machine's internet access. A Mac or Windows machine can be told to block all outbound traffic except what you explicitly allow, and the machine will keep running the model perfectly – because, at this point, nothing about running the model needs the internet. The model file is local, the software is local, and the work happens entirely on the machine.

Chapter 5 goes into the details of how to do this properly and why it matters. For now, note the order: *download first, disconnect second*. You need the internet to fetch the model; you do not need it again after.

Step 6: Hand it over, with a written procedure

The last step is not technical at all. You write down – in plain language, on one page – what the machine is for, how to use it, and what to do when it breaks. Who on the team uses it, what they may and may not put into it, and who to call when the thing won't start. The machine is only useful if the people in the practice can actually use it, and "can actually use it" is a documentation problem, not an engineering problem.

The gotchas that bite beginners

Three specific traps account for most first-week frustration, and all three are avoidable once named.

The offload trap. Some setups let a model "spill over" from fast memory into slow disk-backed memory when it doesn't fit. It looks like it's working — the model loads, the answer comes — but the answer arrives at a crawl, because part of the model is being read from disk in real time. If your model responds in seconds per word instead of words per second, you're offloading, and the fix is a smaller model, not a faster machine.

The "q8 wants 24 GB" surprise. The size jump between q4 and q8 is not a rounding error; it's roughly the difference between 5 GB and 9 GB for a 7–8B model, and it only grows from there. A machine that happily runs a model at q4 can fail to run the *same* model at q8. When you see "out of memory," the first thing to check is whether you're asking for the bigger file.

The update that changed the behavior. We saw this pattern repeatedly: an inference engine or model update lands, and a workflow that worked on Friday misbehaves on Monday. The fix is discipline, not cleverness — update deliberately, test your real work immediately after, and if anything changed, roll back. Chapter 8 returns to this; it's worth flagging here because it's most likely to bite in the first month, right when you're least likely to suspect it.

What actually goes wrong

We have watched a lot of these setups, and the failures are boring and predictable. The most common: **the model doesn't fit.** Someone buys a 16 GB machine and tries to run a 13B model at q8, and it either refuses to load or crawls. The fix is the Chapter 3 rule: size for the model above the one you think you need. The second most common: **the machine "disappears."** A Mac that is supposed to be reachable over the network stops answering because of a FileVault or reboot issue — Chapter 8 is about exactly this, because it is the single most common way a private-AI setup quietly dies.

CHAPTER 5

Chapter 5 – Privacy and Offline Operation

What "no data leaves" means, precisely

The claim you want to be able to make is not "we take privacy seriously." It is: *the machine has no path by which client data can leave the building.* There is a meaningful difference between those two sentences, and it is testable.

A machine is genuinely offline when, if you were to inspect its network traffic over a week of normal use, you would see nothing – or nothing beyond what you explicitly allowed. Getting to that state is not a single setting; it is a handful of deliberate steps, each of which closes one of the quiet ways a machine phones home.

The steps, in order of importance

Cut the network, don't just "not browse." A machine that is simply not being used to browse the web is still connected. Operating systems check for updates, sync services run in the background, and installed software occasionally reports diagnostics. For a compliance story you can defend, you want the network *restricted at the operating-system level* – outbound traffic blocked by default, with specific exceptions only for things you deliberately allow (such as the initial model download, or a controlled off-site backup that you have decided in advance is acceptable). On macOS and Windows this is achievable with the built-in firewall and a small amount of configuration; on macOS

the `Little Snitch`-style or built-in application firewall, and on Windows the built-in firewall rules, will do it.

Disable telemetry and diagnostics. Both major operating systems include "send usage data to improve the product" features, on by default in some configurations. Turn them off. Individually they are small; collectively they are the difference between "no data leaves" and "mostly no data leaves."

Fix the software you installed to not phone home. The inference engine you installed is the software that actually handles your data, so it is the one whose behavior matters most. Check its settings for anything labeled telemetry, crash reporting, update checking, or "share usage data," and disable them. The good engines work fully offline by design; verify that yours does.

Air-gap the truly sensitive machine, if you can. The strongest arrangement — appropriate for the most sensitive practices — is a machine that is *never* connected to the internet after initial setup. Model updates are done by physically moving a downloaded file onto the machine from a separate, trusted computer (a USB drive, or a second machine on your own network). This is more work, but it converts "the machine has no path out" from a configuration claim into a physical fact.

The backup tension, and how to resolve it

There is one place where "offline" and "good practice" genuinely conflict: backups. If the machine is air-gapped and its disk dies, you lose the model and any work on it. The honest resolution is to

decide *in advance* what leaves, and only let that leave. A sensible policy: the model file itself (which is public and contains no client data) can be re-downloaded anytime, so it doesn't need special protection. The things that contain client data — any documents you stored on the machine, any conversation history — should be treated the same way you already treat client files: backed up through your existing, already-vetted, encrypted backup channel, and nothing more.

The principle is simple and worth writing down: **the model is public; your clients' data is not. Keep the two on opposite sides of the line.**

Offline is not a claim you make; it's a test you run

The best thing about "no data leaves the building" is that it's *falsifiable*. You can sit down and verify it. Set up the machine, use it normally for a day, and then look at the network log. If the log shows traffic to addresses you didn't deliberately allow, the machine isn't offline yet — full stop. This is not a philosophical exercise; it's a five-minute check that converts a reassuring sentence into a demonstrated fact.

A useful discipline is to run this check *twice*: once after initial setup, and once again after any change (an update, a new piece of software, a new user's configuration). The second run matters, because that's when the quiet phone-homes creep back in.

One caveat for realism: "offline" and "usable" are in tension in a few specific spots, and you should decide each one explicitly

rather than let it happen. The main spots are: operating-system updates (security patches you probably *do* want), the model download itself (requires the network once), and any legitimate off-site backup you've chosen. The resolution isn't to pretend these don't exist; it's to allow them explicitly, on a schedule you set, through a path you control — and to document that you did. A private AI setup is not about having zero network contact ever; it's about having zero *unexplained* network contact.

A governance lesson from the field: provenance

One more thing falls under "privacy" even though it is easy to file under "a different department." We once built a voice clone locally — synthetic speech of a real, consenting person, generated entirely on our own hardware, with the audio never leaving the building. Every technical component worked correctly. And we realized, after the fact, that **we had never marked the output as synthetic.** The provenance — "this is a clone, not the person" — was never attached to the thing we produced.

The lesson transfers directly to a professional practice. When you generate text or other content with a local model, the custody of the data is only half the compliance story. The other half is *provenance*: knowing, and being able to show, what was human-written and what was machine-generated. For a practice using AI to draft documents, that means a simple, written rule — for example, "machine-generated drafts are labeled as such until a licensed professional has reviewed and accepted them." Private AI keeps the data in; provenance keeps you able to answer for what came out.

Chapter 6 – Trusting What the Machine Says

The reliability problem does not go away when the server goes away

Here is the single most important thing to understand about running your own AI: **making it private fixes who can see the data. It does not fix whether the machine is right.** If anything, moving to a local model can make the reliability problem *more* visible, because you no longer have a vendor's marketing department between you and the model's actual behavior. You are now looking at it directly. This chapter is about what you will see when you look – because we have looked, and measured, and written it all down.

The model stopped saying "no"

This was our first serious lesson, and it is directly relevant to medical, legal, and accounting work. We were tuning a model for a clinical-like task – the kind of task where the model should sometimes *decline to answer* rather than guess. We made a change intended to make the model more decisive, and then measured the result across a set of test questions.

The model's score fell by **11 points – and the entire regression was concentrated in a single "answer bucket."** At first we published that the model had developed a bias toward always answering. That diagnosis was wrong. What had actually

happened was the opposite and subtler: the model had learned to **over-hedge** — to answer everything in a noncommittal, fence-sitting way — and that hedging is what tanked the score.

The lesson for a practice is this: a model does not reliably tell you when it doesn't know. It hedges, or it sounds confident, or it quietly changes the shape of its answer, but it does not tap you on the shoulder and say "I'm not sure." The "I don't know" signal is exactly the thing these models are bad at. So the human in the loop is not a nice-to-have; it is the *actual safety mechanism*, and you should design your workflow assuming the model will sometimes be confidently wrong.

You can't prompt your way to both commitment and abstention

Because "make it say no when unsure" is such an obvious fix, we tried. We tested **nine different fixes** on a frozen model — the kind of thing a practitioner might reasonably try, from prompt wording to specific parameters. Four of them were outright refuted: they didn't do what they claimed. The eventual winner was a single constant that, for clinical work, you set to zero. But the durable finding wasn't the constant. It was this: **you cannot tune a model to be simultaneously maximally decisive and maximally cautious.** Those two goals pull in opposite directions, and no amount of prompt-crafting resolves the tension. You have to choose which failure you prefer — too-eager or too-hesitant — and for professional work, the answer is almost always: prefer hesitance, and put a human in the loop to resolve it.

The benchmark said 18%. Production said 0.07%.

This one is a warning about how you evaluate *any* AI tool, local or not. We built what we thought was a clever "router" — a system that would inspect each incoming request and decide whether to send it to a fast model or a slow, expensive model. Before building it, we estimated from benchmarks that it would matter for about **18%** of requests. Then we looked at real traffic for thirty minutes.

In production, the case the router was designed for occurred on **0.07%** of requests. The router would have been a machine designed for a problem that basically doesn't happen. We retired it before building it.

The transferable lesson: **benchmarks describe a different population than your actual work.** The only evaluation that matters is the one you run on your own real tasks and your own real traffic. When you set up a local model, test it on your documents, your letters, your summaries — not on a public leaderboard.

When the evaluation itself lies

Two more findings round out the picture, and both are about the *measurement* failing rather than the model failing.

In one case, we were running an evaluation harness — the software that scores how well the model did — and it produced a **complete, plausible, fabricated results table on a server error.** Every failure had silently scored **0.5**, which is the "neutral" middle value, and the harness reported a full, professional-looking table

as if nothing had happened. If we hadn't checked the raw logs, we would have believed a model was performing fine when in fact the evaluation had never actually run.

In another case, while assembling training data for a local project, we discovered that **22% of our training set had been invented by another model** — and the invented rows had been assigned the *highest* sample weight, meaning they were teaching the downstream model the wrong thing most strongly.

Both point the same direction. The tools around a model — the evaluation software, the data pipeline — can fail silently and look like success. For a practice, the implication is concrete: **do not trust a score you did not see produced with your own eyes, and spot-check the underlying output, not just the summary.**

Three more ways the system lies to you

The reliability problem isn't only in the model's answers. It's in the machinery around it, and we documented three specific failure modes that all carry the same lesson: *a system can be failing in a way that looks, from a distance, like success.*

One slow request locks out everything. We had a machine serving requests, and we noticed that under load, a single request would return at **1 token per second** — unusably slow — while the *same* model in isolation returned at **42 tokens per second.** The machine wasn't broken. It was *saturated*: one heavy request was monopolizing the work, and every other request queued behind it. The mistake we nearly made was concluding "the model is too

big for the machine," when the real answer was "we're overloading it." For a practice, the practical meaning is: if your assistant is sometimes fast and sometimes glacial, the machine is probably fine and you're asking too much of it at once — serialize the big jobs, or size up.

A safety feature was silently eating answers. We had a content-safety hook that, in the name of being cautious, was *truncating the model's output* — cutting off long answers and code — and marking the result as a clean, successful "stop." The model looked like it had finished; it had been cut off. Again, the tell was that we checked the raw output rather than trusting the summary. This is a direct parallel to a practice that lets a "content filter" silently drop or truncate a document: the filter works, the document is mangled, and nobody notices until a client does.

A health checker was manufacturing the stalls it was meant to detect. We built a monitoring probe that ran periodically to make sure the machine was healthy — and discovered the probe itself was consuming **98% of the logged requests** and a large share of the GPU's time. The monitoring was the load. It's a wry lesson, but it lands for any practice that sets up a tool to watch the tool: make sure the watcher isn't the thing degrading what it watches.

Each of these, again, was caught not by cleverness but by *looking at the actual output and the actual numbers.* There is no substitute for it.

The practical standard

Putting it together, the working standard for a small practice is a sentence you can tape to the monitor: **the model drafts, the human decides.** The model writes the first version; a licensed professional reviews, corrects, and signs. The model is a very fast, very confident junior colleague who occasionally invents things — useful precisely because you treat it that way, and dangerous the moment you stop.

CHAPTER 7

Chapter 7 – What It Actually Costs

The shape of the cost

Local AI has a cost profile that is almost the inverse of cloud AI. Cloud AI is a small, recurring, per-seat expense that never goes away. Local AI is a one-time capital expense (the machine) plus a small, ongoing expense (electricity, and occasional time). Neither is obviously cheaper in the abstract; the answer depends on how many people use it and for how long. This chapter shows the arithmetic plainly, using only numbers you can verify, so you can compute it for your own situation.

The capital cost

Using the prices from Chapter 3, rounded and marked as street prices:

- Entry (Mac mini M4, 32 GB): roughly **$800**
- Mid (GPU PC or Mac mini M4 Pro, 64 GB): roughly **$1,600–$1,900**
- High (Mac Studio M4 Max, 64 GB): roughly **$3,000**

That's the whole capital outlay. There is no per-seat license for the model (open-weight models are free to download and use), and no per-request charge. Once the machine is paid for, it costs the same to run it once a day or a hundred times a day.

The ongoing cost: electricity

This is where local AI wins in a way that surprises people, and the numbers are worth stating precisely.

- A **Mac Studio idles around 10 watts** and draws **under 200 watts** under full inference load.
- A **GPU PC idles around 60–90 watts** and draws **500–700 watts** under sustained load.

Translate that to money. If you run the machine for about four hours a day of actual inference, the GPU PC costs on the order of **$8–12 more per month** in electricity than the Mac does. Both are small numbers in the context of a practice's operating budget — we're talking about a rounding error on the electric bill — but the Mac's advantage is real and it compounds over the life of the machine. It's also quieter and cooler, which matters in an office.

The comparison to a cloud subscription

Here is the honest arithmetic. I am deliberately not going to quote a specific cloud AI seat price, because those prices change constantly and vary by vendor and tier, and this book's rule is not to state a price I can't stand behind. Instead, here is the formula, which you can fill in with the real numbers from any vendor's pricing page today:

> **Total cost of local (3 years) = hardware cost + (electricity per month × 36) + (the value of your own maintenance time).**

> **Total cost of cloud (3 years) = (price per seat per month) × (number of seats) × 36.**

The decision turns on one comparison. A local machine is a fixed, one-time cost that does not grow with headcount and does not go up every time the vendor raises prices. A cloud subscription is a cost that scales linearly with the number of users and is completely outside your control.

The pattern to notice: if you have one or two people using AI for an hour a day, the cloud subscription is cheap and the $800–$3,000 machine can look like overkill. If you have a whole office — six, ten, fifteen people — each pasting documents into an AI assistant, then every additional seat is a recurring charge *forever*, while the local machine's cost is unchanged. The break-even point is specific to your seat count and your vendor's price, and the formula above is exactly how you find it.

A worked example

Let's make it concrete with the entry machine, and a generic cloud seat price you can replace with your vendor's real number.

Suppose a four-person practice. Local setup: a Mac mini M4 with 32 GB, around $800, drawing on the order of 30 watts under use — call the electricity negligible for a machine that size. Cloud setup: four seats at, say, $Y per seat per month. (I'm deliberately not naming a vendor price; you have the actual number on their page as you read this.)

Local, three years: $800, full stop — plus your maintenance time, which Chapter 8 says is a few hours a quarter at most.

Cloud, three years: 4 seats × $Y × 36 months = 144 × $Y.

Set them equal and solve: **144 × $Y = $800 → $Y ≈ $5.50.** In other words, if your per-seat price is above about six dollars a month, the $800 machine pays for itself inside three years for a four-person office — *and* you've solved the custody problem as a free side effect, which is really the point. Run the same arithmetic for a ten-person office and the break-even per seat is under three dollars a month.

The numbers shift with the hardware tier — the $3,000 Mac Studio needs a higher seat price or a longer horizon to pencil out on cost alone — but the shape never changes: the local machine is a fixed cost, and the cloud is a cost that multiplies by headcount forever.

What you should *not* count on saving

For honesty's sake, one cost people imagine local AI saves and it mostly doesn't: **compute efficiency.** We spent a long stretch squeezing efficiency out of our own setups, and the results were genuinely discouraging in places. A model's "thinking" (reasoning) mode cost us **2.05× the tokens** at the median and up to **9.66×** on one extraction task. In a separate text-to-speech pipeline, we found **64% of the compute was being thrown away** to a batching misconfiguration. And a benchmark showed prefill — the

setup phase before the model starts answering — eating **13.4% of total inference time.**

The point of these numbers is not that local AI is inefficient (cloud AI has the same physics). It's that *nobody hands you efficiency for free.* Whether you rent or own, the way you drive the machine matters, and a poorly-driven local machine can waste as much as a cloud subscription. The difference is that when you own it, you can see the waste and fix it — and you're not paying per token while you figure it out.

The cost nobody budgets: utilization

There is one more number from the field that should temper any plan to "buy a big machine and serve the whole office." We ran a GPU cluster for our own work — a serious, multi-machine setup — and then measured how much of the time it was actually doing work.

It was **idle 97.4% of the time.** A **2.6% duty cycle.** And that mattered: we spent effort chasing a 29% throughput improvement, and the honest conclusion was that the improvement was nearly worthless *because the machine is almost never busy.* Most of the time, an AI machine sits there doing nothing, waiting for the next question.

The lesson for a practice is not "don't buy the machine." It is **"buy the machine that matches your actual, measured usage, not the machine that matches your imagination."** A solo practice does not need a Mac Studio. A six-person office does not need a

GPU server. Match the hardware to the real workload, and remember that most of the hours in a day, the thing will be idle – and an idle Mac mini costs almost nothing to have sitting there, while an idle GPU server is a small heater you're paying to run.

CHAPTER 8

Chapter 8 – Keeping It Running

The real cost of local AI is not money. It's attention.

Everything up to this point describes getting a machine working. The harder, less glamorous part is keeping it working over months. This is the true, rarely-advertised cost of running your own AI, and you should enter into it deliberately rather than discovering it at month four when the machine is a paperweight and nobody remembers who was supposed to maintain it.

The good news is that the failures are boring and diagnosable. The bad news is that they require *someone* to notice. Here is what actually happens, drawn from our own field notes.

The machine was up. Everything on it was dead.

This is the canonical local-AI failure, and it is worth reading twice. We had a Mac that was **powered on and answering network pings – and everything meaningful on it had died.** Ninety-three background jobs had been registered in a part of the system that gets torn down when the graphical login session ends. The machine *looked* alive from the outside and was functionally a brick on the inside. The monitoring said "up." The reality was "dead."

The transferable lesson: **"the machine responds" is not the same as "the service works."** Your monitoring has to check the

actual thing — does the model answer a question? — not just whether the box answers a ping.

Four wrong diagnoses of a locked-out Mac

We spent a genuinely frustrating stretch unable to reach a Mac that had been working fine, and we diagnosed it wrong three times before finding the real cause. The real causes, once we found them, were:

1. **FileVault's pre-boot screen was answering our network requests** — so the machine *looked* reachable, but it was actually sitting at a login prompt waiting for a password, not running any of our stuff.
2. **User-level background jobs don't survive a reboot** — on macOS, the jobs that start with your user account are not guaranteed to come back up after a restart unless they're set up to, and ours weren't.
3. **A headless Mac can't grant itself the permission it needs** — the machine required a user to click a dialog box to approve something, and there was no monitor and no user, so it sat there forever waiting for a click that was never coming.

Each of these is, in hindsight, obvious. Each of them cost us real time precisely because we assumed the machine was fine and looked everywhere except the dumb, boring answer.

The transferable lesson for a practice is concrete and valuable: **a private-AI machine needs a monitor, or it needs to be**

configured so it never, ever asks for anything. A headless machine that needs a password, a reboot, or a permission click is a machine that will silently stop working the first weekend you're out of the office.

The maintenance that is actually required

Strip away the horror stories and the real, recurring maintenance is modest. It is:

- **Periodic model updates.** Open-weight models improve over time. A couple of times a year, download the newer version of your model and swap it in. This takes minutes and needs the network (see Chapter 5's air-gap procedure).
- **Software updates, done deliberately.** Update the inference engine occasionally — but do it *when you choose*, not automatically, and re-test your actual work afterward, because Chapter 6 taught us that a changed component can silently change behavior.
- **Backups.** The model is re-downloadable; your documents and settings are not. Back them up through your existing channel.
- **A monthly five-minute check.** Once a month, ask the model a real question. If it answers, the machine is healthy. This single ritual catches 90% of the "silent death" failures described above, because the failure mode of local AI is almost always *quiet*.

The rest of the failure catalogue

Because "keep the machine alive" is the whole hidden cost, it's worth listing the other ways we've seen healthy-looking systems fail, each of which maps to a boring practice rule.

Everything was configured, and nothing took effect. Four separate reasons a configuration edit didn't change behavior — and every one of them was a "the config file you edited isn't the one the program reads" style of problem. The rule: after any change, verify the *behavior* changed, not that you changed a file.

The tool logged success and did nothing. A compression job reported "success" while compressing zero bytes. The success message and the actual work had become disconnected. The rule: a green checkmark means someone *claims* success, not that work happened — spot-check the output.

Twenty-five skills loaded, zero actually worked. An agent setup reported loading 25 capabilities, of which none functioned, because the files were one directory level too deep. The rule: a count of things loaded is not a count of things working.

The error message named the wrong cause. A "429 too many requests" error that was actually a malformed request (a 400) wearing a rate-limit costume. The rule: read the error, then verify it, because error messages are written by fallible humans too.

The watchdog was starved. The process whose job was to make sure everything else was healthy didn't have the resources to run. The rule: the thing you rely on to watch the system is part of the system, and can fail with it.

None of these requires an engineer to fix, but each requires a *person* to notice, and to have the standing permission to spend an hour looking instead of clicking "restart" and hoping. That person is the "named owner" from the next section — and if you don't have one, every item on this list eventually becomes "the machine stopped working and nobody knows why."

Who owns it

The final, non-technical requirement: **one named person owns the machine.** Not "the office manager, whoever that is this year." A specific person, with a written one-page runbook (what it does, how to test it, who to call). This is not a technology decision; it's an operating decision. A private-AI setup is like a company car or a filing cabinet — it only works if someone is accountable for it. The practices that get value from local AI are the ones that assign this ownership up front. The ones that don't, end up with an expensive box in a closet by spring.

CHAPTER 9

Chapter 9 — The Decision Checklist

A go / no-go test you can run in an hour

This chapter is the book condensed into a decision. Work through the questions in order. By the end, the answer should be obvious, and if it isn't, the answer is "not yet."

Part 1: Is this your problem?

1. Do you handle data you are legally or ethically obligated to protect?

If you're a medical, legal, or accounting practice — or any business holding client data under a confidentiality or regulatory obligation — the answer is yes. If you don't hold such data, you can still benefit from local AI, but the *urgency* is lower; the compliance argument is the strongest reason to go local.

2. Do you currently use (or want to use) a cloud AI assistant with real client data?

If the answer is yes, you have a custody problem today, and this book is directly aimed at you. If the answer is "no, we don't use AI at all," then the question is whether the productivity gain is worth the setup — a calculation, not a compliance emergency.

Part 2: Is the workload a match?

3. Is your work mostly *text* – summarizing, drafting, explaining, reformatting documents?

Local models are strong at exactly this. If your work is mostly numeric calculation, live web lookup, or image/video generation, a local model is a weaker fit and you should be honest about that.

4. Is your usage a few people, a few hours a day, with peaks and valleys?

This is the sweet spot. Local AI is best when usage is intermittent (so the machine sits idle and cheap most of the day) and the users are a known, small set (so you don't need a server farm).

5. Can you tolerate "drafts, not final answers"?

If a workflow requires zero human review, no model – local or cloud – is safe for you. If you can build review into the workflow (and for professional work you should), local AI is a fine fit.

Part 3: The money and the owner

6. Run the Chapter 7 arithmetic with your real numbers.

Fill in your seat count and your vendor's per-seat price, and compare against the hardware tiers. The answer is usually decisive one way or the other.

7. Is there a named person willing to own the machine?

No named owner, no local AI. This is a hard requirement, not a nicety.

The verdict

- If you answered **yes to 1 and 2** (regulated data + cloud AI use), you have a live compliance problem, and the cost of the machine is almost certainly justified by the cost of the problem it solves. **Go — and prioritize the offline setup in Chapter 5.**
- If you answered **yes to 1 but no to 2** (regulated data, but not currently using AI), you have time. Read Chapters 6 and 8 before you commit, and start with a Tier 1 machine as a trial. **Go, but deliberately.**
- If you answered **no to 1** (no regulated data), you are weighing pure productivity, and the honest answer may be that a cloud subscription is simpler and fine. **Decide on cost, not on principle.**

When not to do this

Be willing to say no. Local AI is the wrong answer when: your work isn't mostly text; you need many simultaneous users; you need the absolute best model available at any cost; or no one in the practice will take ownership of a machine. In those cases, use cloud AI deliberately and with your eyes open — or don't use AI yet at all. There is no shame in deciding a tool isn't right for you; there is real harm in half-adopting it.

Conclusion

The argument of this book, in three sentences:

A regulated small practice has a specific problem that cloud AI does not solve: the custody of its clients' data. That problem is now solvable with hardware that costs less than a good laptop and fits on a desk. The cost of that solution is not money — it is honesty about what these machines can and cannot do, and a named person willing to keep one running.

Everything else in these pages is the working-out of those three sentences. The memory math in Chapter 3 tells you what to buy. The setup in Chapter 4 tells you how to stand it up. Chapter 5 tells you how to make "no data leaves" a fact instead of a slogan. Chapter 6 tells you why you still need a human in the loop, and gives you the measured reasons to believe it. Chapter 7 tells you what it costs, honestly. Chapter 8 tells you what happens after the honeymoon. And Chapter 9 is the whole thing turned into a checklist.

If you take one thing from this book, take this: **running your own AI is not a hobby project anymore. It is a business decision with a clear answer for a defined set of businesses.** For the medical office, the law firm, and the accounting practice, the answer is increasingly "buy the machine, cut the network, and keep the data where it belongs — inside the building."

Appendix — Where the numbers come from

This book is built from roughly a year of hands-on work running open-weight language models on self-owned hardware, documented in a set of field notes. For readers who want to verify or go deeper, here is the mapping between the claims in this book and the underlying findings. Every number cited above comes from one of these entries; nothing was invented to round out the prose.

On the memory formula (Chapter 3). The bytes-per-parameter figures (q4 ≈ 0.6, q8 ≈ 1.1, fp16 = 2.0), the KV-cache and overhead budgets, and the model-size table are drawn from the hardware guidance notes used to advise buyers, and are consistent with published quantization practice. Hardware tier prices are street prices at the time of writing.

On model reliability (Chapter 6).

- "The model stopped saying no": the 11-point regression concentrated in a single answer bucket, and the corrected "over-hedging" diagnosis.
- "Can't prompt your way to both commitment and abstention": nine fixes tested on frozen logprobs, four refuted, one constant to set to zero for clinical work.
- "The benchmark said 18%, production said 0.07%": a gateway router retired by thirty minutes of real-traffic analysis.

- "Every failure silently scored 0.5": an evaluation framework producing a plausible fabricated results table on a server error.
- "22% of our training set was invented by another model": the fabricated rows carried the highest sample weight.

On utilization and cost (Chapter 7).

- "Our GPU cluster is idle 97.4% of the time": a 2.6% duty cycle, and a throughput upgrade it rendered near-worthless.
- Power-draw figures (Mac Studio ~10 W idle / <200 W load; GPU PC 60–90 W idle / 500–700 W load) and the $8–12/month electricity difference at ~4 hours/day of inference.

On maintenance (Chapter 8).

- "The machine was up. Everything on it was dead": 93 background jobs in a launchd domain that dies with the GUI session.
- "Four wrong diagnoses of a locked-out Mac": FileVault pre-boot answering network requests, user LaunchAgents not surviving reboot, and a headless Mac unable to grant its own permission.
- "We built a voice clone and never marked it as synthetic": the provenance gap on locally generated synthetic speech.

The field notes are published, in full, with their corrections and retractions, on the site that accompanies this book (robotnurse.io).

If a claim in this book and a field note ever disagree, trust the field note — it is the primary source, and this book is the summary.

Glossary

The terms below appear throughout the book. They are defined here as plainly as possible, in the order a reader is likely to encounter them, rather than alphabetically.

Model. The file of numbers that produces the AI's output. A "language model" is trained on a great deal of text and, given a prompt, predicts what text should come next. "Model" and "AI" are used loosely as synonyms in this book.

Parameters. The individual numbers inside a model file. A "7-billion-parameter model" contains roughly 7 billion of them. More parameters generally means a more capable model — and a larger file.

Open-weight. A model whose parameter file is published for anyone to download. Once downloaded, you have a fixed copy that cannot be changed or revoked by the publisher.

Quantization. Storing a model's numbers at lower precision so the file is smaller and needs less memory. "q4" stores each parameter in about 4 bits, "q8" in about 8 bits, "fp16" in 16 bits. Lower precision means a smaller file and, usually, a small quality trade-off.

Inference. The act of running the model to produce an answer. "Training" is building the model; "inference" is using it. When this book says "running the model," it means inference.

Inference engine. The software that loads a model file and runs inference. Ollama and LM Studio are two common ones.

Token. The unit of text a model reads and writes. Roughly, a token is a short fragment of a word. "Tokens per second" is how model speed is measured; a handful of words per second feels responsive, while a word every few seconds does not.

Context window. How much text the model can hold in its working memory at once — the conversation plus the document you pasted. Larger context windows let you feed in longer documents but use more memory.

KV cache. The model's working memory during a conversation. It grows with the length of the conversation and the context window, and it consumes memory beyond the model file itself.

Unified memory. Memory shared between a Mac's processor and graphics hardware. Because it's shared, a Mac can give a large share of its memory to a model — which is why a 64 GB Mac can run models a 16 GB graphics card cannot.

VRAM. Dedicated memory on a graphics card. It's very fast but limited in amount, which is why graphics cards are excellent for small models and cannot run large ones at all.

Offload. The (usually automatic) fallback of putting part of a model into slower, disk-backed memory when it doesn't fit in fast memory. Offloading makes the model run very slowly.

Hallucination. A model producing text that is fluent and confident but factually wrong. The term is slightly misleading — the model isn't imagining so much as statistically generating plausible text — but it's the standard word. This book's position is that hallucination is not a bug to eliminate but a risk to manage with human review.

Telemetry. Data an application silently sends back to its maker about how it's being used. For a private AI setup, telemetry is one of the main things to disable, because it is exactly the kind of quiet "phone home" that violates the "no data leaves" guarantee.

Air-gap. A computer with no network connection at all. The strongest form of offline operation: data physically cannot leave because there is no path out.

Business Associate Agreement (BAA). Under HIPAA, the contract a covered health-care provider must have with a vendor that will handle protected health information. Its absence is a common compliance gap when practices use consumer AI.

Provenance. A record of where something came from — for example, whether a document was written by a human or generated by a machine. This book treats provenance as a compliance obligation, not a technical detail.

Latency. How long you wait for an answer to start or complete. **Throughput** is how much work the machine gets through over time. A system can have good throughput and bad latency, or vice

versa — the number that matters for one person typing questions is latency.

Fine-tuning. Training an existing model further on a specific body of material to make it better at a particular task. Mentioned in passing in this book because it's a way practices can go further; it is not required for the setup described here.