RUNNING

LOCAL LLMs

ON YOUR OWN HARDWARE

PRIVATE BY DESIGN

100% LOCAL

MAXIMUM PERFORMANCE

FRONTIER INTELLIGENCE

MODEL
Llama 3 70B Q4_K_M

CONTEXT
32,768

VRAM USAGE
22.1 / 24 GB

TOKENS / SEC
68.4 tok/s

INFERENCE
Running

STATUS
LOCAL

Your data.
Your rules.

YOHAN J. RODRÍGUEZ

Preface

"The future is already here — it's just not evenly distributed."
William Gibson

For most of the last decade, using a large language model meant sending your text to someone else's computer. That is changing. Open-weight models now run well on hardware you already own — a gaming laptop, a workstation with a single consumer GPU, an Apple Silicon Mac, or even a capable CPU with enough memory. Running these models yourself buys you privacy, predictable cost, offline availability, and a level of control that no hosted API can offer.

This book is a practical, hands-on guide to doing exactly that. It is written for developers, tinkerers, and technically curious professionals who want to download a model, run it on their own machine, serve it to their own applications, and understand the tradeoffs well enough to make good decisions. You do not need a machine-learning background. You do need to be comfortable on a command line.

A word on scope. This book is about *running* open-weight models, not building them: it does not teach how to train a model from scratch, the deep mathematics of transformers, or large-scale production MLOps, and it is not a complete reference for every option of any single tool — those move too fast to pin down on paper. What it does cover is the durable craft of choosing, running, serving, adapting, and operating local models well. Where a tool's exact flags or a model's version numbers matter, treat the book's examples as a starting point to confirm against current documentation, not a frozen specification.

The examples target Linux first — specifically Debian and Ubuntu with NVIDIA GPUs — because that is the most common self-hosting setup and the easiest to write precisely. Where macOS (Apple Silicon and Metal), AMD (ROCm), Windows, or CPU-only paths differ in an important way, a call-out explains what changes. Every command in this book is meant to be typed and run, not just read.

The goal is not to memorize a tool. Tools change every few months. The goal is to build a durable mental model: how an LLM uses memory and compute, how quantization trades quality for footprint, how to pick a model that fits your hardware, and how to diagnose the slow, broken, or out-of-memory runs you will inevitably hit. Once you have that model, the specific commands become easy to adapt.

Yohan J. Rodriguez

Contents

1 | Why Run LLMs Locally, and the Open-Weight Landscape

A large language model is, fundamentally, a large file of numbers and a program that does arithmetic with them. For the first few years of the modern LLM era, almost nobody thought of it that way, because the only practical way to use one of these models was to send your text across the internet to a company that ran the arithmetic for you and sent back the result. That arrangement is convenient, and for many people it is the right choice. But it is not the only choice, and it has not been the only choice for some time. Open-weight models that you can download and run on your own computer now reach a level of quality that was, very recently, available only behind a paid API. A model you can run on a mid-range gaming laptop today would have been considered a serious research artifact a few years ago.

This chapter is about why you might want to run these models yourself, what you gain and what you give up, and what the landscape actually looks like: which models exist, what "open weight" really means, where the files live, and what you can realistically expect to run on the hardware you already own. By the end of the chapter you will have run a real model on your own machine and called it from code, so that the rest of the book has a concrete reference point rather than an abstract one.

1.1 What This Book Means by "Local" and "Your Own Hardware"

Throughout this book, *running a model locally* means that the model weights live on storage you control and the inference computation happens on a processor you control — your CPU, your GPU, or the unified memory of your Apple Silicon Mac. No part of the prompt or the response leaves your machine unless you deliberately send it somewhere. There is no per-token bill, no account, and after the initial download there is no requirement for an internet connection at all.

This is a different thing from "self-hosting" in the cloud sense, where you rent a server from a provider and run software on it. You can absolutely run everything in this book on a rented GPU instance, and some readers will, but the center of gravity here is the machine on your desk or

under it. "Your own hardware" in the title is meant literally: a desktop with a consumer graphics card, a laptop, a Mac, a small home server, or a workstation. The techniques scale up to rented and on-premise servers without changes, and the final chapters discuss scaling explicitly, but the assumption that you own and physically control the machine is what makes the privacy and cost arguments in this chapter real rather than aspirational.

1.1.1 The Shape of a Local Inference Setup

Every local setup, no matter which tool you use, has the same three pieces. First, there are the *weights*: a file (or a few files) containing the model's learned parameters, typically downloaded once and reused forever. Second, there is an *inference engine*: a program that loads the weights into memory and runs the math to turn a prompt into output tokens. Third, there is an *interface*: a chat prompt in your terminal, a local web UI, or — most importantly for developers — a small HTTP server that speaks a familiar API so your own programs can call the model. Chapters 5 through 7 install and operate each of these pieces in detail. For now, the important idea is that local inference is not exotic infrastructure; it is a file, a program that reads the file, and a way to talk to that program.

That three-part picture — weights, engine, interface — is the minimum, and it is all you need to chat with a model. A more capable setup that serves applications, grounds answers in your own documents, or drives tools layers a few more pieces on top of it, and it helps to see the whole stack once as a map: not to build it all now, but to know what each layer does and where this book covers it. Table 1.1 names the layers from the model outward; the lower ones are present in every setup, while the upper ones are added only when your use calls for them, and most readers start with the first few and grow into the rest.

Layer	**What it does**	**Covered in**
Model weights	The learned parameters — the model itself	Chapter 4
Tokenizer	Splits text into the tokens the model reads	Chapter 2
Inference engine	Loads the weights and runs the math	Chapter 6
API server	Exposes an OpenAI-compatible HTTP endpoint	Chapter 10
UI / front-end	A chat window for people	Chapter 6
Agent harness	Plans, calls tools, runs multi-step tasks	Chapter 9
Tools	Functions the model can call (search, code)	Chapter 9
Memory / RAG	Grounds answers in your own documents	Chapter 11
Evaluation	Measures whether a model is good enough	Chapter 4
Operations	Serving, security, monitoring, maintenance	Chapter 15

Table 1.1: The layers of a local LLM system, from the model outward. The lower layers are in every setup; the upper layers are added only as your use requires — a map of the book as much as of the stack.

You will rarely touch all ten layers at once, and you should not try to. The value of the map is

that when a later chapter discusses an "API server" or an "agent harness," you will know where it sits in the whole — and that a complete local AI system is these few comprehensible layers stacked on the simple file-plus-program core, not a monolith you must master in one go.

1.2 A Short History of How Local LLMs Became Practical

It is worth understanding how we arrived at the present moment, because the history explains why the tools look the way they do and why "run it yourself" went from impossible to routine in a remarkably short time.

The first widely known large language models were closed by default. The models that kicked off the public conversation were accessible only through hosted APIs or web interfaces; the weights were not released, and even if they had been, the hardware required to run them was far beyond anything an individual owned. The implicit assumption of that era was that using a capable model meant renting access to someone else's data center, and for a while that assumption was simply correct.

The turning point was the release of capable *open-weight* base models by major labs, beginning with Meta's first LLaMA models. Suddenly the weights existed in the open. They were still difficult to run — they assumed datacenter GPUs and full-precision math — but the raw material was now in the community's hands, and a wave of practical engineering followed almost immediately. Two developments in particular turned "the weights exist" into "you can actually run this on your laptop."

The first was *aggressive quantization*. Researchers and engineers showed that the weights could be stored at much lower numerical precision — four bits per weight instead of sixteen — with surprisingly little loss of quality. That single change cut the memory footprint by roughly a factor of four, which is the difference between a model that needs a server and one that fits on a consumer graphics card. Chapter 2 explains why this works and Chapter 8 explores it in depth, but its historical importance is hard to overstate: quantization is what made local inference fit in local memory.

The second was the appearance of *efficient inference engines built for ordinary hardware*, the most influential of which was llama.cpp. Where the original code assumed Python and datacenter GPUs, these new engines were written in portable, optimized C and C++ that ran well on CPUs, on consumer GPUs, and on Apple Silicon, and they defined practical file formats — culminating in GGUF — for distributing quantized models. Around the same time, the technique of *instruction tuning* turned raw text-completion base models into helpful assistants that follow instructions and hold conversations, which is what most people actually want.

Put those pieces together — open weights, four-bit quantization, efficient cross-platform engines, and instruction tuning — and the result is the world this book is written for, where a genuinely useful assistant runs on hardware you already own. The pace has not slowed. New

model families appear constantly, each generation more capable at a given size than the last, and a steady stream of efficiency work keeps pushing what fits on modest hardware. The specific models named in this book will be superseded; the structure that made them runnable will not.

1.3 Five Reasons to Run Models Locally

People come to local models for different reasons, and the reason you care about shapes the choices you make later — which model, which quantization, which engine. It is worth being honest with yourself about which of the following motivates you, because some of them push toward small fast models and some push toward large capable ones.

Figure 1.1: Five key reasons to run language models locally.

1.3.1 Privacy and Data Control

The most common reason, and the one with the clearest payoff, is that your data never leaves your machine. When you call a hosted API, your prompt — which may contain source code, customer records, medical notes, unpublished writing, legal documents, or internal strategy — travels to a third party. Reputable providers have policies about retention and training, and those policies are often good, but a policy is a promise, not a physical guarantee. A local model offers the physical guarantee: the bytes are processed by a chip you own and are never transmitted.

This matters in concrete, boring, everyday ways long before it matters in dramatic ones. A lawyer running a model over privileged documents, a clinician summarizing patient notes, a company processing data covered by contractual confidentiality, an engineer pasting a proprietary stack trace, a writer who simply does not want their draft sitting in someone else's logs — all of them get a categorical benefit from local inference that no API-side policy can match.

It is worth dwelling on how often this applies, because the instinct to think of privacy as an edge case is misleading. Most professionally interesting text is confidential by default. Source code is usually proprietary. Customer records are almost always governed by privacy law or contract. Internal documents, financial figures, hiring notes, security findings, unreleased research, and personal correspondence are all things you would hesitate to paste into a stranger's web form — and a hosted API is, at the level of data flow, exactly that. Local inference does not ask you to make a careful judgment about whether a given piece of text is sensitive enough to keep off the network; it removes the question entirely by keeping everything off the network. For many people that simplification — never having to decide — is the single most freeing thing about running models themselves. If you operate under GDPR, HIPAA, or similar regimes, "the data was never sent anywhere" is a dramatically simpler compliance story than "the data was sent to a processor with whom we have a data-processing agreement."

It is worth being precise about what local inference does and does not protect, because "local" is a property of a correctly configured system, not a magic word. A properly local setup — weights on your disk, inference on your chip, and the server listening only on `localhost` — guarantees that prompts and responses are not transmitted to any third party. That is the strong, categorical protection, and it is the one most people are after. What local inference does *not* do by itself is protect you from mistakes in your own configuration. If you bind the inference server to a public network interface without authentication, anyone who can reach that port can use your model and see traffic that passes through any logging you have enabled. If you build a pipeline that, say, sends part of a prompt to a hosted service for one step, that step is no longer local. And local inference says nothing about the physical security of the machine itself: an attacker with access to your disk has access to whatever you stored there. The lesson, which Chapter 15 develops into concrete practice, is that local inference gives you a genuinely private *foundation*, and your configuration choices determine whether you keep that privacy or accidentally give it away.

1.3.2 Cost That Does Not Scale With Usage

Hosted APIs charge per token. For light or occasional use, this is cheap and entirely reasonable. But three situations flip the economics toward local inference. The first is high volume: classifying millions of records, generating embeddings for a large document corpus, processing a continuous stream of events, or running an agent that makes thousands of calls in a loop. The second is long-running development: when you are iterating on prompts and pipelines all day, every test

run costs money, and the meter creates a subtle pressure to experiment less. The third is simple predictability: a local model has a fixed cost — the hardware and the electricity — and that cost does not change whether you make ten calls a day or ten million.

Table 1.2 contrasts the two cost models. The point is not that local is always cheaper; for a few thousand tokens a month, a hosted API is cheaper than the electricity to keep a GPU warm. The point is that the cost curves have completely different shapes, and at high or sustained volume the local curve wins decisively because its marginal cost per token approaches zero.

A worked example makes the crossover concrete, even with rough numbers. Suppose a GPU capable of running a useful model costs a few hundred dollars and draws, say, two hundred watts while busy. The hardware is a one-time cost you can amortize over years; the electricity, at typical residential rates, comes to a small number of cents per hour of heavy use. Against that, a hosted API charges per million tokens, and a moderately heavy workflow — an agent looping over a task, a nightly batch job over thousands of documents, or a team iterating on prompts all day — can easily consume tens of millions of tokens in a month. At hosted per-token rates, that monthly bill recurs every month, forever, and grows with usage; the local machine's cost is mostly the up-front purchase plus a few dollars of electricity, regardless of how many tokens you push through it. The exact break-even depends on your rates and volume, but the structure is unavoidable: below some usage threshold the API is cheaper, and above it the owned hardware is cheaper, and the threshold is low enough that anyone running models all day crosses it quickly. The deeper point is not the dollar figure but the *shape*: owning the hardware converts a variable, usage-coupled, open-ended operating expense into a fixed, predictable one, and that predictability is itself worth something to anyone trying to budget or to experiment freely without watching a meter.

Aspect	**Hosted API**	**Local model**
Upfront cost	None	Hardware (one time)
Per-token cost	Metered, ongoing	Effectively zero
Cost at high volume	Grows linearly	Flat
Cost predictability	Variable	Fixed
Idle cost	Zero	Electricity / depreciation
Best for	Light, bursty use	Heavy, sustained use

Table 1.2: Hosted versus local cost models. The curves have different shapes; volume decides the winner.

1.3.3 Latency, Availability, and Offline Use

A local model answers without a network round trip. For interactive use the difference is usually small, but for two cases it is decisive. The first is genuine offline operation: on a plane, in a secure facility, in a remote location, or during an internet outage, a local model keeps working when

a hosted one simply does not exist. The second is tight loops: when a program calls a model many times in sequence — an agent, a batch pipeline, a retrieval system reranking results — the network latency of each call adds up, and removing it can change a workflow from sluggish to snappy.

There is also an availability argument that has nothing to do with the network. A hosted model can change underneath you: the provider can deprecate the version you depend on, alter its behavior with an update, rate-limit you during a traffic spike, or discontinue the service. A model file on your disk does exactly what it did yesterday, forever. For anything you need to depend on long-term, that stability is worth a great deal.

Consider a concrete case that combines both effects. An engineer builds an internal tool that runs a model over every incoming support ticket to classify and route it. With a hosted model, that tool inherits two ongoing risks: a network or provider outage stops ticket routing entirely, and a silent model update can shift the classifications without warning, quietly degrading the routing until someone notices. The same tool built on a pinned local model has neither risk: it routes tickets during an internet outage, and its behavior is frozen until the engineer deliberately changes it. Neither property is about peak capability; both are simply consequences of owning the thing you depend on, and for production systems that ownership is frequently the deciding factor regardless of any quality comparison.

1.3.4 Control, Customization, and Reproducibility

When you run the model, you control everything about how it runs. You choose the exact model version and pin it. You set sampling parameters precisely. You can fine-tune the model on your own data (Chapter 12), constrain its output to a strict grammar or JSON schema (Chapter 9), inspect or modify the system prompt, and run experiments that a hosted API would never expose. You can also reproduce results exactly: the same weights, the same parameters, and a fixed seed produce the same output, which matters enormously for research, for testing, and for any setting where you must explain why the system produced what it did.

This control extends to behavior that hosted providers deliberately restrict. Local models can be run without the additional refusal layers and content filters that hosted services wrap around their models. That capability is a double-edged tool — it is what makes local models suitable for legitimate work on security research, fiction, medicine, and other domains where hosted filters get in the way, and it is also a responsibility, because the guardrails are now yours to provide where they are needed.

Reproducibility deserves a sentence of its own because it is easy to undervalue until it bites you. With a hosted model, the version you tested against can change, which means a prompt that worked perfectly in development can behave differently in production weeks later through no change of your own. A pinned local model removes that variable entirely: given the same weights, the same parameters, and a fixed seed, the output is identical run to run. For automated tests, for

scientific work that must be reproducible, for auditing why a system produced a particular answer, and for debugging in general, this determinism turns the model from a shifting dependency into a fixed, inspectable component of your system.

1.3.5 Learning and Understanding

Finally, there is the reason that this book quietly assumes you share at least a little: running models yourself is how you actually understand them. When the model is a remote black box, concepts like quantization, context length, the KV cache, and tokens per second are abstract. When you are the one watching VRAM fill up, choosing a quantization level, and measuring throughput, those concepts become concrete and intuitive. Even if you ultimately use hosted APIs for production, the understanding you build by running models locally makes you better at using any model, anywhere. The mental model is portable; the specific commands are not.

This learning dividend compounds in unexpected directions. Once you have watched a model's output degrade as you push the quantization too low, you understand in your bones why a hosted provider's "fast" tier might feel slightly worse than its "quality" tier. Once you have managed a context window that overflowed and saw the model forget the start of a conversation, you understand why long hosted chats sometimes lose the thread. Once you have measured the difference between the first-token delay and the steady generation rate, you can reason about why an assistant feels sluggish to start but then races. None of this knowledge is available from the outside of a black box, and all of it makes you a sharper, more effective user of every model you touch. Running models locally is, among other things, the best available course in how language models actually behave.

1.4 What You Give Up

A book that only listed the advantages would be selling something. Local inference involves real tradeoffs, and you will make better decisions if you hold them clearly in mind.

The largest tradeoff is raw capability at the top end. The biggest hosted frontier models are larger than anything you can run on consumer hardware, and on the hardest reasoning, coding, and knowledge tasks they remain ahead of what a single GPU can run. The gap has narrowed remarkably — a good open model on a single high-end consumer GPU is genuinely useful for the large majority of everyday tasks — but if your work depends on the absolute frontier of capability, a local model may not match it, and you should test rather than assume.

The second tradeoff is effort. A hosted API is one HTTP call away. A local model requires you to choose hardware, install drivers, pick a model and a quantization, manage disk space, and occasionally debug an out-of-memory error or a driver mismatch. This book exists precisely to make that effort small and predictable, but it is not zero, and it is ongoing in the sense that the ecosystem moves quickly.

The third tradeoff is that you become the operator. There is no one to email when something breaks, no autoscaling when you need more throughput, and no security team patching the service. You own uptime, updates, backups, and security. For a personal setup this is trivial; for anything others depend on, it is a real responsibility that the final chapters of this book take seriously.

A fourth tradeoff is subtler and worth naming because it surprises people: the ecosystem moves fast enough that a local setup is never quite "finished." New models arrive that are better than what you have, engine updates change flags and defaults, and a GPU upgrade can mean reinstalling drivers and rebuilding tools. None of this is hard once you have the mental model this book builds, and you are under no obligation to chase every release — a working setup keeps working. But the field rewards a little ongoing attention, and it is honest to set the expectation that local inference is a small, pleasant hobby-or-craft to maintain rather than a one-time installation you never think about again. The later chapters on performance, scaling, and maintenance are written precisely to make that ongoing attention efficient rather than burdensome.

Dimension	**Local**	**Hosted**
Privacy	Data stays on your machine	Data sent to provider
Cost at scale	Flat, hardware-bound	Linear, per token
Peak capability	Bounded by your hardware	Frontier models available
Setup effort	Real, one-time + ongoing	Minimal
Operations	You own it	Provider owns it
Offline	Works	Does not exist
Customization	Full (fine-tune, sampling, grammars)	Limited to exposed knobs

Table 1.3: A balanced view. The right answer is often "both", used for different jobs.

The honest conclusion, summarized in Table 1.3, is that local and hosted models are complementary rather than competing. Many practitioners use local models for the bulk of their work — the private, the high-volume, the iterative — and reach for a frontier hosted model for the occasional task that genuinely needs it. This book teaches the local half thoroughly precisely because it is the half that is less obvious and more rewarding to learn.

1.4.1 A Decision Framework: Local, Hosted, or Both

Rather than treat the choice as an identity — "I am a local-models person" — it is more useful to make it per workload, by asking a short series of questions about the task in front of you. The answers usually point clearly one way.

The first question is about *data sensitivity*. If the data is private, regulated, or contractually confidential, that alone is often enough to favor local, because no quality advantage offsets a confidentiality breach. If the data is public or non-sensitive, this question does not constrain you and you move on to the others.

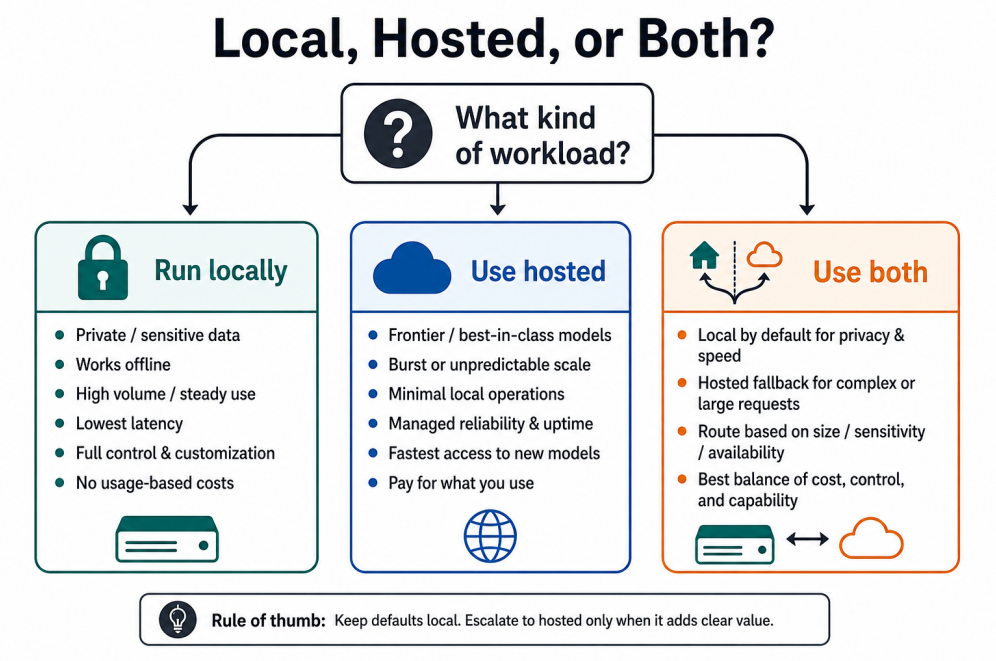

Figure 1.2: Choose local, hosted, or hybrid model use by workload.

The second question is about *volume and frequency*. If the task runs constantly, in tight loops, or over large batches, the flat cost curve of local inference and the absence of per-call latency favor local strongly. If the task is occasional and light, a hosted API's zero setup and zero idle cost may win.

The third question is about *required capability*. If the task is squarely in the strong band of mid-sized models — everyday writing, summarization, routine coding, extraction, grounded question answering — a local model handles it well. If the task is at the genuine frontier of difficulty, test whether a local model is good enough before assuming, and keep a frontier hosted model available for the cases where it is not.

The fourth question is about *operational constraints*. If the task must run offline, must be reproducible, or must be pinned against silent change, local is the natural fit. If you need someone else to own uptime and scaling and you are willing to pay for that, hosted is reasonable.

In practice these questions rarely conflict, and when they do, data sensitivity usually dominates. The mature posture most experienced practitioners arrive at is "both, deliberately": a capable local model as the default workhorse for the private, the high-volume, and the routine, with a frontier hosted model held in reserve for the occasional task that genuinely earns it. Because both speak the same API, as the hands-on section will show, routing a workload to one or the other is often a single line of configuration rather than an architectural decision. The goal of this book is to make the local option a genuine, well-understood choice in that framework — not to insist it is the only one.

1.5 A Vocabulary for the Rest of the Book

A handful of terms recur on nearly every page from here on. You will meet each one again in proper depth — most of them in Chapter 2 — but a brief, plain-language definition now gives you anchors so the next few sections read smoothly rather than cryptically. Treat this as a glossary to glance back at, not something to memorize.

A *token* is the unit of text a model reads and writes. It is usually a word fragment rather than a whole word: "running" might be one token or split into "run" and "ning", and a rough rule of thumb is that a token is about four characters of English. Models think in tokens, are priced in tokens by hosted providers, and generate one token at a time.

A *parameter* (or *weight*) is one of the learned numbers inside the model. "7B" means roughly seven billion of them. The parameter count is the single biggest determinant of how much memory a model needs and how capable it tends to be.

Quantization is the technique of storing each parameter at lower numerical precision — commonly four bits instead of sixteen — to shrink the model in memory. It is the reason large models fit on small hardware, and at modern bit depths it costs surprisingly little quality for the memory it saves — a trade-off, not a free lunch, and one Chapter 8 measures precisely.

The *context window* is the maximum amount of text, measured in tokens, the model can consider at once: your prompt plus the conversation history plus the response all share this budget. Larger context costs more memory, for reasons Chapter 2 makes precise.

VRAM is the memory on a graphics card, and it is usually the binding constraint on what you can run, because the model's weights must fit in it for fast inference. *Unified memory* is the Apple Silicon equivalent, shared between CPU and GPU.

Inference is the act of running a trained model to produce output — as opposed to *training*, which is creating or modifying the model in the first place. An *inference engine* is the program that does this: it loads the weights and runs the math. *Tokens per second* is the headline speed metric, measuring how fast the model generates text.

The *KV cache* is a block of memory the engine uses to avoid redoing work as it generates each new token; it grows with the length of the conversation and is a major consumer of memory during long chats. Finally, *GGUF* is the file format most local engines use to store quantized models — when you download a model to run with llama.cpp or Ollama, you are usually downloading a GGUF file.

With those anchors in place, the landscape and the hands-on sections that follow will read as concrete rather than jargon-laden.

1.6 “Open Weight” Versus “Open Source”

You will see the terms “open source” and “open weight” used loosely and often interchangeably. The distinction matters, both legally and practically, so it is worth getting right at the start.

A model is *open weight* when the trained parameters — the weights — are made available for download so that anyone can run the model. This is the property that makes local inference possible. Almost every model in this book is open weight, and for the purpose of running a model on your own machine, open weight is the property you need.

A model is *open source* in the fuller sense when, in addition to the weights, the training code, the training data, and the methodology are released under a license that permits use, study, modification, and redistribution. Truly open-source-everything models exist, but many of the most popular “open” models are open weight without being fully open source: you can download and run and often fine-tune them, but you do not get the training data, and the license attaches conditions.

1.6.1 Licenses You Will Actually Encounter

Licenses range from genuinely permissive to meaningfully restricted, and a few minutes reading the license of a model you depend on can save you a serious problem later. The common categories are these.

Permissive open-source licenses such as Apache 2.0 and MIT place essentially no restrictions on use: you may run the model commercially, modify it, and redistribute it freely. Several major model families use Apache 2.0, and for commercial work these are the easiest to reason about.

Community or custom licenses are released by some labs alongside their models. These typically permit broad commercial and research use but add conditions — for example, an acceptable-use policy, a requirement to display attribution, or a clause that restricts use above a very large user threshold aimed only at the largest companies. For the overwhelming majority of readers these conditions are irrelevant in practice, but you should read them rather than assume.

Research-only or non-commercial licenses permit experimentation and study but forbid commercial deployment. Some strong models fall in this category. They are excellent for learning and prototyping and unsuitable for a product without separate permission.

The practical rule is simple: before you build something that matters on a particular model, open its model card and read the license section. It takes two minutes and it is the difference between a safe foundation and an unpleasant surprise. Throughout this book, when a specific model is used in an example, the choice favors permissively licensed models so that nothing you learn here is encumbered.

1.7 The Open-Weight Model Landscape

The set of available models changes month to month, and any specific list in a book will age. What does not age as quickly is the set of *families* — the lineages produced by particular labs, each with its own character, license posture, and range of sizes. Knowing the families lets you navigate new releases as they appear, because a new model is almost always a new entry in a family you already understand.

1.7.1 The Major Families

Table 1.4 sketches the landscape at a level that should remain useful even as specific version numbers advance. Treat the sizes as "the kinds of sizes this family ships" rather than an exact catalog.

Family	Typical sizes	Character / notes
Llama (Meta)	1B–70B+	Broad, well-supported, huge ecosystem
Mistral / Mixtral	7B, 8x7B (MoE)	Efficient; strong small and MoE models
Qwen (Alibaba)	0.5B–72B+	Strong multilingual, coding, and math
Gemma (Google)	2B–27B	Compact and efficient; permissive terms
Phi (Microsoft)	3B–14B	Small models punching above their weight
DeepSeek	7B–large MoE	Strong coding and reasoning lineages
Command-R (Cohere)	35B+	RAG- and tool-use-oriented
Yi, InternLM, others	varies	Capable additional lineages worth knowing

Table 1.4: Open-weight model families. New releases are usually new entries in these lineages.

Two structural ideas in that table will recur throughout the book. The first is the *size in parameters*, measured in billions and written like "7B" or "70B". Larger generally means more capable and slower and more memory-hungry; Chapter 4 turns this rough intuition into actual memory arithmetic. The second is the *mixture-of-experts* (MoE) architecture, written like "8x7B", in which the model contains many expert sub-networks but only activates a few per token. MoE models can be very capable relative to the compute they use per token, at the cost of needing memory for all the experts at once. Chapter 2 explains both ideas in enough depth to reason about them.

A short profile of each family helps you read new releases as they appear. *Llama*, from Meta, is the most widely supported lineage and the de facto reference: it ships across a wide range of sizes, the tooling supports it first, and an enormous community of fine-tunes and quantizations builds on it. If you are unsure where to start, a current Llama instruct model is rarely a wrong choice. *Mistral* and its mixture-of-experts sibling *Mixtral* made their name on efficiency, delivering strong quality at small sizes and popularizing accessible MoE models under permissive licensing. *Qwen*, from Alibaba, spans an unusually wide size range and is particularly strong at multilingual

text, coding, and mathematics, with dedicated coder variants worth knowing. *Gemma*, from Google, focuses on compact, efficient models that run well on modest hardware. *Phi*, from Microsoft, pursues the "small but smart" thesis, training small models on carefully curated data so they punch well above their parameter count. *DeepSeek* produced influential coding and reasoning lineages, including large mixture-of-experts models. *Command-R*, from Cohere, is tuned specifically for retrieval-augmented generation and tool use, which makes it relevant to Chapter 11. Beyond these, lineages such as Yi and InternLM are capable and worth a look. You do not need to track every release; you need to recognize the family a release belongs to, because that tells you its license posture, its rough character, and which tools will support it on day one.

1.7.2 Base, Instruct, and Chat Variants

Within a family, a given size usually ships in more than one variant, and choosing the wrong one is a common beginner mistake. A *base* model is trained only to predict the next token over text; it is a powerful text completer but it does not naturally follow instructions or hold a conversation. An *instruct* or *chat* model has been further trained to follow instructions and respond in a dialogue format. For almost everything in this book — chatting, answering questions, building assistants, powering applications — you want the instruct or chat variant. Reach for a base model only when you specifically want raw completion or you intend to fine-tune it yourself. When a model is offered as both `Llama-3.2-3B` and `Llama-3.2-3B-Instruct`, the instruct variant is the one you almost certainly want.

1.8 Where the Models Live

Open-weight models are distributed through a small number of well-known channels, and knowing them makes finding and downloading a model straightforward.

Hugging Face is the dominant hub. It hosts the original weights published by the labs, as well as countless community re-uploads in formats convenient for local inference — most importantly the GGUF files used by llama.cpp and Ollama, and the GPTQ, AWQ, and EXL2 files used by GPU-focused engines. A model on Hugging Face lives in a *repository* identified by an owner and a name, like `meta-llama/Llama-3.2-3B-Instruct`, and its *model card* (the repository's README) contains the license, the intended use, the prompt format, and often benchmark numbers. The hub has a command-line tool that downloads either an entire repository or a single file, which matters because you usually want just one quantized file rather than the whole multi-format repository.

The Ollama library is a curated, simplified catalog. Instead of choosing a specific file, you pull a named model like `llama3.2:3b` and Ollama selects a sensible quantization and handles the download and storage for you. This is the easiest on-ramp, which is why the hands-on section below uses it, and Chapter 6 explains exactly what it is doing underneath.

Quantizer repositories are community accounts on Hugging Face that specialize in converting new models into GGUF and other local-friendly formats, often within hours of a model's release. When a model you want is not yet available as a GGUF from its original publisher, a reputable quantizer's repository is usually where you will find it. Chapter 8 discusses how to read the quantization labels on these files and choose among them.

A note on safety that Chapter 15 develops fully: model files are data, but some legacy formats can execute code when loaded. The modern formats this book uses — `safetensors` for raw weights and `GGUF` for quantized weights — are designed to be safe to load, whereas older Python `pickle`-based checkpoints can run arbitrary code. Prefer the safe formats, and download from reputable sources, exactly as you would with any other software.

1.8.1 Reading a Model's Name and Card

Model names look intimidating until you learn that they are simply several facts concatenated together. Decoding them is a small skill that pays off constantly, because the name usually tells you everything you need to decide whether a file fits your hardware. Consider a typical file name you might download: `Llama-3.2-3B-Instruct-Q4_K_M.gguf`. Read left to right, it says: the *family* is Llama; the *version* is 3.2; the *size* is 3 billion parameters; the *variant* is Instruct (so it follows instructions, as opposed to a bare base model); the *quantization* is `Q4_K_M`, meaning roughly four bits per weight using the "K-quant medium" scheme; and the *format* is GGUF, so it is meant for llama.cpp or Ollama. From that one string you already know it will run in a few gigabytes of memory, that it will chat sensibly, and which engines can load it.

The quantization suffix is the part newcomers find most opaque, and Chapter 8 is devoted to it, but a working heuristic is enough for now: a higher number means more bits, more memory, and slightly higher quality, so `Q8_0` is large and near-lossless, `Q4_K_M` is the popular balance of quality and size, and the very small quants trade noticeable quality for the ability to fit on tight hardware. When in doubt, a `Q4_K_M` or `Q5_K_M` file is the sensible default, which is why community quantizers label them as recommended.

The *model card* — the README on the model's Hugging Face page — is the companion skill. It is worth a two-minute read before you commit to a model, and you are looking for five things: the *license* and whether it permits your use; the *intended use and variant* so you pull instruct rather than base; the *prompt or chat template* the model expects, because using the wrong template is the single most common cause of disappointing output (Chapter 9 returns to this); the *context length* the model supports; and any *benchmark numbers*, which are useful for comparison but should be read skeptically, since models are sometimes tuned to look good on popular benchmarks in ways that do not translate to your tasks. Treat the card as a spec sheet: skim it deliberately, and you will avoid the majority of beginner frustrations.

1.9 What Can You Realistically Run?

The single question every newcomer asks is: "What can my machine actually run?" The honest short answer is that the binding constraint is almost always *memory* — specifically VRAM on a GPU, or unified memory on Apple Silicon, or system RAM for CPU inference. The model's weights must fit in memory, plus some overhead for the context. Chapter 3 teaches you to read your hardware precisely and Chapter 4 gives you the exact arithmetic, but Table 1.5 offers an orienting preview so you can set expectations now.

Memory available	Comfortable model size (4-bit)	Example use
8 GB VRAM	7B–8B	Chat, coding help, summarization
12 GB VRAM	8B–13B	The same, with more headroom
16 GB VRAM	13B–14B	Stronger reasoning and coding
24 GB VRAM	30B–34B	Noticeably more capable
48 GB VRAM	70B (tight)	Near-frontier on a single card
32–64 GB RAM (CPU)	7B–14B, slowly	No GPU needed; a few tokens/sec
128 GB RAM (CPU)	30B–70B, very slowly	Large models via patience and swap
Apple Silicon (unified)	scales with RAM	7B–70B depending on the Mac

Table 1.5: A rough guide to model size by available memory, assuming 4-bit quantization. Chapter 4 makes this exact.

Two clarifications keep this table honest. First, the sizes assume *4-bit quantization*, the compression technique introduced in Chapter 2 and detailed in Chapter 8, which is why an "8B" model fits comfortably in 8 GB rather than needing the 16 GB its full-precision weights would require. Second, CPU inference works — you do not need a GPU to follow most of this book — but it is far slower than GPU inference, producing a few tokens per second rather than dozens, because, as Chapter 2 explains, generation is bound by memory bandwidth and a GPU has vastly more of it. If you have only a CPU, you can still run everything here; you will simply choose smaller models and exercise more patience.

1.10 Choosing Where to Run: Deployment Shapes

"Your own hardware" is not one thing, and the shape of machine you use changes what is comfortable. It is worth recognizing the common shapes so you can see which one you have — or which one to aim for — and what each is good at.

A *laptop without a discrete GPU* is the most constrained but also the most common starting point. It runs small models (roughly up to 7B–8B at 4-bit) on the CPU, slowly but usably, and it is perfect for learning, for light personal use, and for development against a small model before deploying against a larger one elsewhere. Everything in this book's early chapters runs here.

A *desktop or workstation with a consumer GPU* is the sweet spot for most readers. A single card with 8 to 24 GB of VRAM runs models from 7B up to around 34B at 4-bit at comfortable speeds, which covers the overwhelming majority of real work. This is the configuration the book treats as its primary target, and it is where the price-to-capability ratio is best.

An *Apple Silicon Mac* is a distinctive and increasingly popular option because its unified memory is shared between CPU and GPU, so a Mac with a lot of RAM can run surprisingly large models without a discrete graphics card. The tooling supports Apple's Metal backend well, and a well-specified Mac is one of the easiest ways to run large models quietly and efficiently.

A *dedicated home server* — a desktop you leave running, possibly with one or more GPUs — turns local inference into an always-available service for your household or team, reachable from your other devices over the local network. This is where the serving and security chapters (10 and 15) earn their keep, because the moment a model is reachable from more than one machine, authentication and network binding stop being optional.

Finally, a *rented cloud GPU* is not "your own hardware" in the strict sense, but everything in this book runs on one unchanged, and it is the pragmatic choice when you need a bigger GPU than you own for a bounded task — a fine-tuning run, a benchmark, an occasional large-model session — without buying the card. The tradeoff is that the privacy guarantee weakens to "as private as your trust in the provider," which is a different and weaker promise than truly local inference.

Most readers will use more than one of these over time: learn on a laptop, settle on a desktop with a GPU, perhaps add a Mac or a home server, and rent a big GPU for the occasional heavy job. The techniques are the same across all of them; only the model sizes and the speeds change.

1.11 What People Actually Build With Local Models

The reasons and the hardware are abstract until you see what they add up to. It helps to know the concrete things people build on local models, both to spark ideas and to calibrate which model size and which later chapters matter for your goals.

The most common use is a *private assistant for everyday knowledge work*: drafting and editing text, summarizing documents, answering questions, brainstorming, and explaining things, all without sending the material anywhere. A 7B-to-14B instruct model handles this well, and the appeal is precisely that the content — a manuscript, a contract, a performance review — never leaves the machine.

A close second is a *coding assistant*. Local code models power editor integrations that complete code, explain unfamiliar functions, write tests, and answer questions about a private codebase. Because the code stays local, this is attractive for proprietary or sensitive projects where pasting source into a hosted service is unacceptable. Chapter 10 shows how to connect a local model to the tools that expect an OpenAI-style endpoint, including IDE assistants.

Document question-answering over your own files — retrieval-augmented generation — is one of the highest-value local applications, and Chapter 11 builds one end to end. You point the system at your documents, and it answers questions grounded in them: internal wikis, research libraries, legal or medical archives, personal notes. Keeping both the documents and the model local means the entire knowledge base stays private.

At larger scale, local models excel at *bulk data processing*: classifying, extracting, tagging, and transforming large volumes of records where a per-token bill would be prohibitive and where throughput matters more than the absolute peak of capability. This is where the flat cost curve of local inference, and the high-throughput engines of Chapter 6, pay off most clearly.

People also build *agents and automation* — programs that call the model repeatedly in a loop to plan, use tools, and act — where removing network latency and per-call cost from a tight loop changes what is practical. And many run local models for *offline and edge* scenarios: on a boat, in a clinic without reliable connectivity, in a secure facility, or simply on a laptop on a plane. Finally, some go a step further and *fine-tune* a model on their own data to specialize it for a domain or a voice, which Chapter 12 covers.

You do not need to pick one of these now. The point is that the same handful of building blocks — a model, an engine, an API, optionally retrieval or fine-tuning — recombine into all of them, which is why the book teaches the blocks rather than a single recipe.

1.12 What Local Models Are Good and Bad At

Setting expectations honestly now prevents disappointment later. A capable local model in the 7B-to-30B range, which is what most readers will run, is genuinely strong at a wide band of tasks: everyday writing and editing, summarization, explanation and tutoring, routine coding and debugging help, structured extraction and classification, translation for major languages, and grounded question-answering when paired with retrieval. For these, a good local model is not a compromise; it is simply the right tool, and often a faster and more private one than a hosted alternative.

The same models have real limits, and knowing them lets you design around them. They have a fixed knowledge cutoff and no awareness of recent events unless you supply that information in the prompt — which is one of the strongest arguments for the retrieval techniques in Chapter 11. They are weaker than the largest frontier models on the hardest multi-step reasoning, on long and intricate coding tasks, and on niche or highly specialized knowledge. Like all language models, they can state false things confidently, so any use where correctness matters needs verification, grounding, or a human in the loop. And smaller models in particular have a shorter effective attention span: they handle a focused task better than a sprawling, underspecified one.

The practical implication runs through the whole book. You get the most from a local model by matching it to tasks in its strong band, by grounding it in real information rather than relying

on its memory, by controlling its output with the parameters and constraints of Chapter 9, and by reserving the occasional frontier-hard task for a larger model — local or hosted — when one is genuinely warranted. Local inference is not about pretending a 7B model is a frontier system; it is about discovering how much excellent, private, inexpensive work the model in front of you can actually do.

1.13 The Tooling You Will Meet in This Book

Before the hands-on section, it helps to have a map of the tools the rest of the book covers, so that names you encounter feel familiar rather than arbitrary. You do not need to install any of these yet — Chapters 5 and 6 do that carefully — but a one-paragraph orientation to each will make the later chapters read more smoothly.

llama.cpp is the foundational open-source inference engine for running models efficiently on CPUs, GPUs, and Apple Silicon. It defines the GGUF file format, pioneered much of the practical quantization work, and ships a server that many other tools wrap. It is the engine you reach for when you want control and broad hardware support; Chapter 6 builds it from source.

Ollama is a higher-level tool built on top of the same foundations. It manages model downloads, storage, and a background service with a single friendly command-line interface, and it exposes both its own API and an OpenAI-compatible one. It is the easiest starting point and the one the next section uses.

vLLM is a high-throughput serving engine aimed at GPUs. Its design — particularly a technique called PagedAttention — lets it serve many concurrent requests efficiently, which makes it the right tool when you are serving a model to multiple users or a high-volume pipeline rather than chatting with it yourself. Chapters 6 and 10 cover it.

LM Studio and Jan are graphical desktop applications for people who prefer a polished UI to a terminal, and *text-generation-webui* is a feature-rich web interface popular for experimentation. *Hugging Face TGI* and *TabbyAPI*/*ExLlamaV2* round out the server-side options. Chapter 6 compares all of them so you can choose deliberately rather than by accident.

Table 1.6 previews these tools so the names have shape before Chapter 6 installs them. Do not try to choose now; the table is a map, not a decision.

Tool	**Best for**	**Interface**
Ollama	Easiest start; personal use	CLI + API
llama.cpp	Control; broad hardware; CPU/Metal	CLI + server
vLLM	High throughput; many users	GPU server (API)
LM Studio / Jan	GUI desktop experience	Graphical app
text-generation-webui	Experimentation; many features	Web UI
TGI / TabbyAPI	Server-side GPU serving	API server

Table 1.6: The engines at a glance. Chapter 6 installs each and explains when to choose it.

The throughline is that all of these tools, however different their interfaces, do the same fundamental thing: load weights, run inference, and expose a way to talk to the model — increasingly through the same OpenAI-compatible API, which is why a single client can talk to any of them. That convergence is what makes the next section's quick taste so portable, and it is why learning one engine well transfers almost entirely to the others.

1.14 A Ten-Minute Taste: Run a Model Right Now

Enough orientation. The fastest way to make everything above concrete is to run a model, and the fastest path to that is Ollama, because it handles installation, download, storage, and serving in a handful of commands. This section is a genuine first run, not a toy; the same model you start here is one you might keep using. Chapters 5 through 7 revisit every step with full explanations and alternatives, so do not worry if some details are not yet clear — the goal here is a working result and an early intuition.

1.14.1 Install, Pull, and Chat

On Linux or macOS, the listing below installs Ollama, downloads a small but genuinely capable 3-billion- parameter instruct model (roughly two gigabytes at 4-bit), and drops you into an interactive chat. The download is the only slow part, and it happens once.

Listing 1.1: Install Ollama, pull a small model, and start chatting

```
#!/usr/bin/env bash
# A complete first run: install Ollama, pull a small model, and chat.
# Works on Linux (Debian/Ubuntu) and macOS. Takes about five minutes
# plus the model download.

# 1. Install Ollama (Linux). On macOS, download the app from ollama.com.
curl -fsSL https://ollama.com/install.sh | sh

# 2. Pull a small, capable instruct model (about 2 GB at 4-bit).
ollama pull llama3.2:3b

# 3. Start an interactive chat. Type a question, then /bye to exit.
ollama run llama3.2:3b
```

When the chat starts, ask it something — "Explain what a GPU is to a ten-year-old," or "Write a Python function that reverses a string" — and watch the answer stream back, generated entirely on your own machine. Type `/bye` to leave the chat. You have now run a real LLM locally. Everything else in this book is refinement: bigger models, faster inference, better control, and serving the model to your own software.

Platform notes. On Windows, install the Ollama desktop application from the project's website rather than using the shell installer; the commands afterward are identical in a terminal. On a Mac with Apple Silicon, the same installer works and the model runs on the GPU through

Metal automatically. If you have an NVIDIA GPU on Linux and the model seems to run on the CPU (you will learn to spot this in Chapter 7), the drivers are not yet set up — Chapter 5 fixes that, and the model still works in the meantime, just more slowly.

1.14.2 Call It From the Command Line

A chat prompt is only the surface. The moment Ollama is running, it exposes an HTTP API on `localhost:11434`, which is what makes a local model useful to your programs rather than just to you. The next listing calls that API directly with `curl` and prints just the reply.

Listing 1.2: Call the local model over its native HTTP API

```
#!/usr/bin/env bash
# Ollama exposes an HTTP API on localhost:11434 as soon as it is running.
# This is a normal local web service: no internet, no account, no key.

# Native Ollama chat endpoint (streaming disabled for a clean one-shot reply).
curl -s http://localhost:11434/api/chat -d '{
  "model": "llama3.2:3b",
  "messages": [
    { "role": "user", "content": "In one sentence, what is a token?" }
  ],
  "stream": false
}' | python3 -c "import sys, json; print(json.load(sys.stdin)['message']['content'])"
```

There is no internet involved in that call, no API key that means anything, and no account. It is an ordinary local web service, and you can call it from any language that can make an HTTP request.

1.14.3 Speak the OpenAI API

The detail that makes local models drop-in useful for existing software is that most engines, Ollama included, also expose an *OpenAI-compatible* API. Any tool or library written to talk to OpenAI can be pointed at your local server by changing one setting — the base URL — and supplying any placeholder key. The following listing makes the same request through the OpenAI-compatible endpoint.

Listing 1.3: The same model through an OpenAI-compatible endpoint

```
#!/usr/bin/env bash
# Ollama also speaks the OpenAI Chat Completions API at /v1.
# Any tool that talks to OpenAI can be pointed here by changing the base URL.
# The API key is required by the protocol but ignored locally; any string works.

curl -s http://localhost:11434/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer not-needed-locally" \
  -d '{
    "model": "llama3.2:3b",
    "messages": [
      { "role": "user", "content": "Name three reasons to run an LLM locally." }
```

```
    ]
  }' | python3 -m json.tool
```

Because the protocol matches, the official client libraries work unchanged. The Python example below uses the real OpenAI client library with nothing altered except the base URL, which means the large ecosystem of tools built around that API — including many IDE assistants and agent frameworks — can run against your local model.

Listing 1.4: The official OpenAI Python client, pointed at localhost

```python
# The official OpenAI Python client works unchanged against a local server.
# Only the base_url changes; the api_key is a placeholder accepted locally.
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:11434/v1",
    api_key="not-needed-locally",
)

response = client.chat.completions.create(
    model="llama3.2:3b",
    messages=[
        {"role": "system", "content": "You are a concise assistant."},
        {"role": "user", "content": "Explain what 'open weight' means in one paragraph."},
    ],
)

print(response.choices[0].message.content)
```

This single fact — that local engines speak the same API as the dominant hosted one — is why self-hosting is practical rather than academic. You can develop against a local model and deploy against a hosted one, or the reverse, or mix them, with the same code. Chapter 10 builds this into a proper service.

1.14.4 Go Beyond the Curated Catalog

Ollama's catalog is convenient but curated. The wider world of models lives on Hugging Face, and you will often want a specific model or a specific quantization that the catalog does not carry. The hub's command-line tool downloads exactly the file you ask for. The listing below fetches a single 4-bit GGUF file — the format Chapter 8 explains — into a local `models` directory, ready to hand to llama.cpp in Chapter 7. The first line installs that tool with `pip`; on recent Debian and Ubuntu this belongs inside a Python virtual environment (Chapter 5 sets one up), since those systems now refuse a `pip install` into the system Python.

Listing 1.5: Download one specific quantized model file from Hugging Face

```
#!/usr/bin/env bash
# Most open-weight models are distributed on Hugging Face. The hub CLI
# downloads a single quantized GGUF file you can hand directly to llama.cpp.

pip install -U "huggingface_hub[cli]"

```

```
# Download one specific 4-bit GGUF file (not the whole repository) into ./models.
huggingface-cli download \
  bartowski/Llama-3.2-3B-Instruct-GGUF \
  Llama-3.2-3B-Instruct-Q4_K_M.gguf \
  --local-dir ./models
```

Finally, it is worth knowing how to take stock of what you have accumulated, because models are large and disk fills up. The last listing lists the models Ollama manages, reports the on-disk size of its model store, and finds any loose GGUF files you have downloaded by hand.

Listing 1.6: See what you have downloaded and how much disk it uses

```
#!/usr/bin/env bash
# Take stock of what you have pulled and how much disk it uses.

# Models Ollama manages, with sizes.
ollama list

# Raw on-disk footprint of the Ollama model store (Linux default path).
du -sh ~/.ollama/models 2>/dev/null

# Any GGUF files you downloaded manually.
find . -name '*.gguf' -printf '%s\t%p\n' 2>/dev/null | sort -rn
```

If you ran these listings, you now have a working local model, two ways to talk to it from the shell, a client-library example, a manually downloaded model file, and a way to keep track of your disk. That is the entire shape of the book in miniature.

1.14.5 Stream the Output and Hold a Conversation

Two refinements turn the one-shot calls above into something that feels like a real assistant. The first is *streaming*: instead of waiting for the whole answer and printing it at once, you can print each token the instant it is generated, which is what makes chat interfaces feel responsive. The next listing requests a streamed response and lets the chunks print as they arrive.

Listing 1.7: Stream tokens as they are generated

```
#!/usr/bin/env bash
# Stream tokens as they are generated, the way a chat UI does, using the
# OpenAI-compatible endpoint. The -N flag tells curl not to buffer, so each
# server-sent-event chunk prints the moment it arrives.

curl -N -s http://localhost:11434/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer not-needed-locally" \
  -d '{
    "model": "llama3.2:3b",
    "stream": true,
    "messages": [
      { "role": "user", "content": "Count slowly from one to five." }
    ]
  }'
```

The second refinement is *memory across turns*. A model has no inherent memory; a multi-turn conversation works only because the program resends the accumulated history with each request. The short Python loop below makes that explicit: it keeps a list of messages, appends each user input and each model reply, and sends the whole list every turn. This tiny pattern is the seed of every chat application, and it runs entirely against your local model.

Listing 1.8: A multi-turn chat loop that maintains history locally

```python
# A minimal multi-turn chat that keeps conversation history locally.
# Each turn appends the user message and the model reply, so the model
# sees the whole conversation. History lives only in this process.
from openai import OpenAI

client = OpenAI(base_url="http://localhost:11434/v1", api_key="local")
messages = [{"role": "system", "content": "You are a helpful, concise assistant."}]

print("Local chat. Type 'quit' to exit.")
while True:
    user = input("you> ").strip()
    if user.lower() in {"quit", "exit"}:
        break
    messages.append({"role": "user", "content": user})

    reply = client.chat.completions.create(model="llama3.2:3b", messages=messages)
    answer = reply.choices[0].message.content
    print(f"bot> {answer}\n")

    # Append the reply so the next turn has full context.
    messages.append({"role": "assistant", "content": answer})
```

Notice that nothing in this loop is specific to local models — it is ordinary client code. That is the recurring theme: once a local engine speaks the OpenAI API, the application code you already know works without modification.

1.14.6 Customize a Model Without Retraining It

You do not have to accept a model's default behavior. Ollama lets you bake a system prompt and default sampling parameters into a named variant with a small file called a *Modelfile*, producing a customized model that still shares the underlying weights on disk. The listing below creates a terse, low-temperature variant and runs a one-off query against it.

Listing 1.9: Bake a system prompt and parameters into a custom model

```
#!/usr/bin/env bash
# Ollama lets you bake a system prompt and default parameters into a named
# model with a small "Modelfile". This creates a customized variant without
# touching the underlying weights.

cat > Modelfile <<'MODELFILE'
FROM llama3.2:3b

# Lower temperature for more focused, deterministic answers.
PARAMETER temperature 0.3
```

```
PARAMETER top_p 0.9

# A persistent system prompt baked into the model.
SYSTEM """
You are a terse technical assistant. Answer in at most three sentences.
Prefer concrete commands and examples over prose.
"""
MODELFILE

# Build the customized model and run it.
ollama create terse-llama -f Modelfile
ollama run terse-llama "How do I list files by size in a directory?"
```

This is a preview of a much larger theme. Chapter 9 explains every parameter shown here — `temperature`, `top_p`, and the system prompt — and Chapter 6 explains Modelfiles in full. The point for now is that local models are not take-it-or-leave-it black boxes; their behavior is yours to shape, from a one-line parameter tweak all the way to the fine-tuning of Chapter 12.

1.14.7 Feel the Size-Speed Tradeoff

One last experiment makes the book's central hardware theme tangible. Pull both a small model and a larger one, ask each the same question, and watch two things: the larger model's answer is usually a little better, and it generates noticeably more slowly and uses more memory. The `-verbose` flag makes Ollama print timing information, including the generation rate in tokens per second, so you can measure the difference rather than guess at it.

Listing 1.10: Compare a small and a larger model on the same prompt

```
#!/usr/bin/env bash
# Feel the size-speed tradeoff for yourself. Pull a small and a larger model,
# then ask each the same question with --verbose so Ollama prints timing,
# including the eval rate in tokens per second.

ollama pull llama3.2:3b      # small: fast, light on memory
ollama pull llama3.1:8b      # larger: a bit slower, more capable

PROMPT="Explain the difference between a process and a thread in two sentences."

echo "=== 3B ==="
ollama run --verbose llama3.2:3b "$PROMPT"

echo "=== 8B ==="
ollama run --verbose llama3.1:8b "$PROMPT"
```

What you observe here is the tension that every subsequent chapter helps you navigate: bigger models are more capable but slower and hungrier for memory, smaller models are faster and lighter but less capable, and the right choice depends on the task and the hardware. If the larger model runs much more slowly than you expected, or fails to load, that is your hardware telling you something — and Chapters 3 and 4 teach you to read that signal precisely and choose accordingly. There is no universally correct model size; there is only the right size for a given job

on a given machine, and developing a feel for that judgment is one of the quiet skills this book aims to build.

1.15 Staying Current Without Getting Overwhelmed

The pace of this field is genuinely fast, and newcomers often feel they are perpetually behind. They are not, and a little perspective helps. New models appear constantly, but, as this chapter has argued, they are almost always new entries in families you already understand, distributed in formats you already know, and runnable with tools you already have. A new release is rarely a reason to relearn anything; it is usually just a better file to drop into the same setup.

A sustainable way to keep up is to follow the *families and the tools* rather than every individual release. When a model family you use publishes a new generation, that is worth your attention; the dozens of intermediate fine-tunes and re-quantizations usually are not, unless one solves a specific problem you have. Likewise, the major inference engines evolve, but their core concepts — the ones this book teaches — are stable, and an engine's release notes will tell you what actually changed.

The community around local models is large, active, and generous, and it congregates in predictable places: the model hubs where weights are published, the issue trackers and discussions of the major engines, and the forums and aggregators where practitioners compare notes on what runs well on what hardware. You do not need to live in these spaces, but knowing they exist means that when you hit an unusual problem — a model that will not load, a quantization that behaves oddly, a new GPU with patchy support — someone has very likely hit it before you and written down the answer. The practical posture is to learn the durable ideas deeply, hold the specific tools loosely, and treat the community as a reference you consult rather than a feed you must follow.

1.16 How to Use This Book

The chapters are ordered to build on one another, and a first read straight through is the intended path. That said, the parts are designed to be useful as references afterward, and readers with different goals can take different routes.

If your goal is simply to *use* a local model well, Part I (foundations), Part II (hardware and models), and Part III (environment and engines) give you everything you need, and you can treat Part IV as a menu. If your goal is to *build* something on top of local models — an application, a pipeline, an assistant — Chapters 9 through 11 on generation control, API serving, and retrieval are the heart of the book for you. If you are chasing *performance* or running *large* models on constrained hardware, Part V is written for you, though it assumes the vocabulary the earlier

chapters establish. And if you intend to *adapt* a model to your own data, Chapter 12 on fine-tuning is the destination, with everything before it as preparation.

Two conventions run throughout. First, every command is meant to be run; the listings are not illustrations but instructions, and the accompanying source files are in the book's code archive. Second, platform differences appear as call-outs labeled for macOS, AMD, Windows, or CPU, so that the main text can stay concrete about the Linux-plus-NVIDIA path without leaving other readers behind.

Table 1.7 is the map of the journey. Each chapter assumes the vocabulary of the ones before it, but the table lets you see the whole route at a glance and jump deliberately once you have the foundations.

Chapter	**What you will be able to do**
1. Why local	Decide when local makes sense; run a first model
2. Under the hood	Reason about memory and speed from first principles
3. Your hardware	Read your machine and estimate what it can run
4. Choosing models	Match a model and quantization to your hardware
5. Environment	Install drivers, CUDA/ROCm, Python, containers cleanly
6. Engines	Install and choose among llama.cpp, Ollama, vLLM, and more
7. First model	Run, inspect logs, and fix common first-run errors
8. Quantization	Pick bit depths confidently; quantize a model yourself
9. Prompting & sampling	Control output with templates, parameters, and grammars
10. Serving as APIs	Expose a model as a reliable local service
11. RAG & embeddings	Ground a model in your own documents, offline
12. Fine-tuning	Adapt a model to your data with LoRA/QLoRA
13. Performance	Measure and improve tokens/sec and latency
14. Scaling	Run bigger models with multi-GPU and CPU offload
15. Security & ops	Keep a deployment private, safe, and healthy

Table 1.7: The book at a glance: each chapter's practical payoff.

1.16.1 Conventions Used in This Book

A few conventions are worth stating once so the chapters can stay focused. Commands are shown in monospaced listings and are meant to be run as written; where a command needs elevated privileges it uses `sudo`, and where a value is a placeholder you should substitute — a path, a model name, a token — the surrounding text says so. Inline references to commands, files, flags, and model names appear in a monospaced style, like `nvidia-smi` or `Q4_K_M`, so they are easy to pick out from prose. Longer code and configuration — shell scripts, Python, YAML, JSON — appears in numbered listings, and every listing in the book is also provided as a file in the accompanying source archive so you can run it without retyping.

Because the book is Linux-first, the main text assumes a Debian- or Ubuntu-style system with an NVIDIA GPU unless it says otherwise, and it uses the package and driver conventions of that

world. When another platform needs a different command or has a meaningful caveat, a short **platform note** calls it out for macOS and Apple Silicon, for AMD GPUs and ROCm, for Windows (native or through WSL2), or for CPU-only setups. These notes are deliberately brief; their job is to keep you unblocked, not to mirror every instruction four times. Finally, the specific model names and version numbers in examples are illustrative snapshots of a fast-moving field — the techniques around them are what endure, and the text flags when a choice is a matter of current fashion rather than lasting principle.

1.17 Common Misconceptions to Leave Behind

A handful of mistaken beliefs cause most of the early frustration with local models, and naming them now saves pain later.

"Local models are toys compared to hosted ones." This was true a few years ago and is no longer true for everyday work. A good open model on a single high-end consumer GPU handles the large majority of real tasks well. The frontier gap persists only at the hardest end, and even there it narrows steadily.

"I need a top-tier GPU to start." You do not. A model runs on an 8 GB card, on an Apple Silicon Mac, or on a CPU with enough RAM. You choose model size to fit your hardware, not the other way around, and the ten-minute taste above runs comfortably on very modest machines.

"Bigger is always better." Bigger is more capable per token and slower and more memory-hungry. A smaller model that responds instantly and fits in memory is frequently the better practical choice, especially for high-volume or interactive work. Matching the model to the job beats maximizing parameters.

"Quantization ruins quality." Modern 4-bit and 5-bit quantization costs surprisingly little quality for an enormous reduction in memory, which is exactly why it is the default throughout this book. Chapter 8 measures the tradeoff so you can see for yourself rather than taking it on faith.

"Running a model locally means my data could still leak." With a properly local setup — weights on your disk, inference on your chip, the server bound to localhost — it does not. The leaks that exist come from configuration mistakes like exposing the server to the network without authentication, which Chapter 15 addresses directly. The default local setup is private by construction.

"Setting this up requires deep machine-learning knowledge." It does not. Running a model is a systems and tooling task — installing software, managing files and memory, calling an API — not a research task. You need to be comfortable on a command line; you do not need to understand backpropagation. This book teaches the small amount of theory that actually informs practical decisions and skips the rest.

"The tools change so fast that learning them is pointless." The specific commands do move quickly, which is exactly why this book emphasizes the durable mental model underneath them. Once

you understand memory limits, quantization, context, and the shape of an inference engine, adapting to a new tool or a new model is a matter of minutes, because they are all variations on the same handful of ideas.

1.18 Frequently Asked First Questions

A few questions come up so consistently from newcomers that answering them here, before the detailed chapters, saves a lot of early confusion.

Do I need an internet connection to use a local model? Only once, to download the model and the tools. After that, inference is fully offline. This is one of local inference's defining advantages.

Do I need a powerful GPU? No. A GPU makes inference much faster, but models run on CPUs and on Apple Silicon, and you choose the model size to fit whatever you have. The ten-minute taste in this chapter runs on very modest machines. Chapter 3 helps you understand exactly what your hardware can do.

Is this legal, and is it free? Running open-weight models is legal and the models themselves are free to download; what varies is the *license* governing how you may use them, especially commercially, which is why reading the model card's license takes priority before you build on a model. The software in this book is open source.

Will a local model be as good as the hosted assistant I am used to? For a great deal of everyday work, yes or close enough; for the hardest reasoning and coding tasks, the largest hosted models are still ahead. The honest answer is to test on your own tasks rather than assume in either direction.

How much disk space will this take? Each model is a download measured in gigabytes — a small model is a couple of gigabytes, a large one can be tens of gigabytes — and they accumulate, so disk planning (covered in Chapter 5) matters more than newcomers expect.

Which tool should I start with? Ollama, exactly as in this chapter, because it handles the most for you. As your needs grow — more control, more speed, more concurrency — Chapter 6 introduces llama.cpp, vLLM, and the rest, and explains when to graduate to each.

Can I use this for my company's data? That is one of the strongest reasons to run locally, because the data never leaves your control. Mind the license for commercial use, and mind the configuration and security guidance in Chapter 15 if the model will be reachable beyond your own machine.

1.19 A Note on Responsible Use

Running models yourself comes with a measure of responsibility that the hosted world partly handled for you, and it would be dishonest to celebrate the freedom without naming the obligation. Local models can be run without the content filters and refusal layers that hosted services apply, which is genuinely valuable for legitimate work that those filters obstruct — security

research, medicine, fiction, sensitive but lawful topics — but it also means the judgment about what to generate and how to use it is now entirely yours. The same independence that protects your privacy also removes a backstop.

Three realities deserve to be held in mind from the start. First, language models can be confidently wrong: they generate plausible text, not verified truth, and a local model is no more immune to this than a hosted one. Any use where accuracy matters needs verification, grounding in real sources (Chapter 11), or a human reviewer — this is a property of the technology, not a bug you can configure away. Second, models reflect biases present in their training data, and running one locally does not neutralize that; if you build something that affects people, you remain responsible for evaluating it for fairness and for the harms it could cause. Third, "you can" is not "you may": the absence of a technical guardrail does not change the legal, ethical, and contractual limits on how you use a tool, and the responsibility for staying within them sits with you.

None of this is a reason for hesitation; it is a reason for intention. The overwhelming majority of local- model use is mundane and beneficial — private assistance, honest work done more cheaply and with better confidentiality. The point of naming the responsibility is simply that ownership is the theme of this entire book, and responsibility is the part of ownership it would be irresponsible to omit. You are taking the wheel; this paragraph is the reminder that the wheel comes with it.

1.20 A Readiness Checklist Before Chapter 2

You do not need anything installed to understand the next chapter, but a few small steps now will make the rest of the book far more rewarding, because you will be able to try everything as you read rather than only reading about it. None of these takes long, and the ten-minute taste above already covered the core of it.

- **Run the ten-minute taste.** Install Ollama, pull a small model, and chat with it once. This single step makes every later concept concrete.
- **Make one API call.** Confirm you can reach the model over HTTP from `curl` or a few lines of code. This is the foundation of everything in Part IV.
- **Locate your hardware facts.** Note roughly how much RAM your machine has, whether it has a discrete GPU and how much VRAM, or whether it is an Apple Silicon Mac and how much unified memory. Chapter 3 turns these numbers into capability; having them ready helps.
- **Check your disk.** Confirm you have at least tens of gigabytes free for models. They add up faster than you expect.

- **Bookmark a model hub.** Know where models live so that when a chapter names one, you can find it and read its card.

If you have done these, you are fully equipped for the rest of the book. If not, you can still read on profitably and set them up when a chapter asks you to; the path is designed to be forgiving either way.

1.21 Recap and What Is Next

This chapter made the case for local inference and mapped the territory. Local means the weights are on your storage and the computation is on your chip, with nothing leaving your machine. The reasons to want this — privacy, cost that does not scale with use, latency and offline operation, control and reproducibility, and understanding — pull in somewhat different directions, and knowing which one drives you will guide your later choices. The tradeoffs are real: the frontier of capability still lives in the largest hosted models, and you become the operator of your own system. "Open weight" is the property that makes local inference possible, and it is distinct from full open source; reading a model's license is a two-minute habit worth keeping. The landscape is best understood as a set of families that ship in base and instruct variants across a range of sizes, distributed mainly through Hugging Face and curated catalogs like Ollama's. What you can run is bound mostly by memory, and a modest machine runs more than newcomers expect. And, as the ten-minute taste demonstrated, the practical experience of running a model and calling it through an OpenAI-compatible API is already within reach in a handful of commands.

1.21.1 Key Takeaways

- **Local means owned.** The weights live on your storage and inference runs on your chip; the inference itself makes no network call — a physical default rather than a policy promise. Keeping data fully local still means disabling tool telemetry and minding logs, the operator's job Chapter 15 details.
- **Five motivations, different pulls.** Privacy, cost-at-scale, latency and offline operation, control and reproducibility, and understanding. Know which one drives you; it guides every later choice.
- **The tradeoffs are real.** The largest hosted models still lead at the frontier, and you become the operator of your own system. Local and hosted are complementary, not rivals.
- **"Open weight" enables local inference** and is distinct from full open source. Read a model's license before you build on it; it takes two minutes.

- **Think in families and variants.** New releases are new entries in known lineages; pick the instruct or chat variant for almost everything.
- **Memory is the binding constraint,** and quantization is what makes large models fit small hardware. A modest machine runs more than newcomers expect.
- **Decode model names and read model cards.** The name encodes family, size, variant, and quantization; the card carries the license, template, and context length you need.
- **You already ran one.** A model plus an OpenAI-compatible API is the whole shape of the book, and it is reachable in a handful of commands.

1.21.2 Where to Go Next

The next chapter opens the hood. To choose models and hardware well, and to understand why generation is slow on a CPU and why context costs memory, you need a working mental model of how transformer inference actually uses compute and memory. Chapter 2 builds exactly that model — tokens, attention, the KV cache, prefill versus decode, and precision — with no more mathematics than the decisions require. If you have not yet run the ten-minute taste, do it before moving on: the rest of the book lands far better when you have a running model in front of you to experiment with as you read.