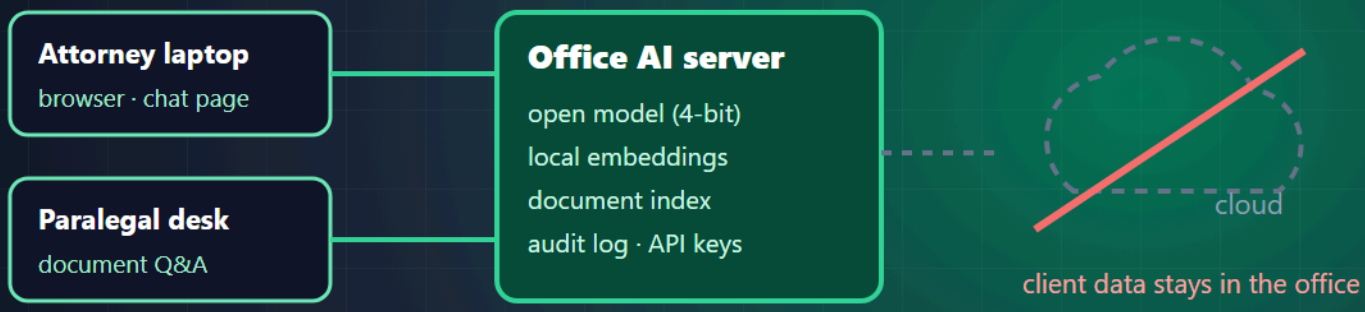
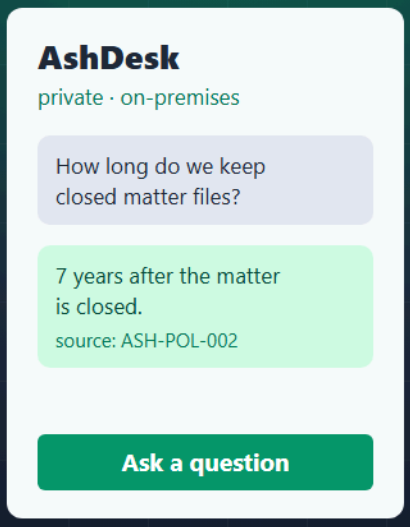



```
# memory for weights
params x bits / 8
8e9 x 4.5 / 8 = 4.50 GB
# KV cache, 8k tokens
131,072 B x 8,192 = 1.07 GB
```

```
POST /v1/chat/completions
{"model": "ashdesk-chat"}
```



A COMPLETE, PRACTICAL GUIDE

Run LLMs Locally

Private AI on your own hardware with llama.cpp, Ollama, and Python

Hardware sizing · Quantization · GGUF · Local APIs · Embeddings
Document Q&A · Tool calling · Office server · Security · Monitoring

SIZE · RUN · BUILD · DEPLOY · SECURE

THENEXBOOKS

Run LLMs Locally

Private AI on Your Own Hardware with llama.cpp, Ollama, and Python

From hardware sizing to a secure, multi-user office assistant, built step by step

First Edition

THENEXBOOKS

Run LLMs Locally
Private AI on Your Own Hardware with llama.cpp, Ollama, and Python

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Open models, runtimes, and libraries change quickly; command names, flags, default values, file formats, and license terms can differ between versions. Always check the documentation and license of the exact versions you use. Nothing in this book is legal advice; consult a qualified professional about your own obligations when you process confidential or personal data.

Python is a trademark of the Python Software Foundation. PyTorch, Hugging Face, Transformers, FastAPI, Ollama, NVIDIA, CUDA, Apple, Windows, Linux, and other product names mentioned in this book are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these companies or projects. Open models are referred to by their public repository names for identification only; each model is subject to its own license.

Ashcombe Legal, the law firm used as the running example in this book, is fictional, as are its people, clients, matters, documents, policies, and all sample data. The sample policies are invented for teaching and are not templates of legal requirements in any jurisdiction. Email addresses use reserved example domains.

Preface

Language models have become everyday tools, but many organizations cannot simply paste their work into a cloud service. A law firm holds privileged client communications. A clinic holds health records. An engineering company holds designs it has promised to keep secret. For them, the question is not whether AI is useful but whether it can be used without sending confidential data somewhere they do not control.

Open models make a different answer possible. You can download a capable model, run it on a laptop, a workstation, or a server in your own office, and keep every prompt and every document inside your own network. The trade-offs are real: smaller models make more mistakes, hardware costs money, and someone has to install, secure, monitor, and update the system. This book shows you how to make those trade-offs with your eyes open.

We start with the arithmetic that decides what you can run at all: how many bytes a model needs, what quantization saves, and why memory bandwidth limits speed. We then work through the main local runtimes, build our own OpenAI-compatible API in Python, and grow it into a private assistant that answers questions from firm documents, calls tools, and serves a whole office with authentication, logging, and monitoring. Every step is explained, and the Python code in the book was run on an ordinary CPU-only computer.

Who This Book Is For

- **Developers** who know basic Python and want to build applications on open models that run on their own hardware.
- **IT and operations staff** asked to provide AI tools to colleagues without sending confidential data to outside services.
- **Technical leads at small and mid-size organizations** (law firms, clinics, consultancies, engineering offices) who must decide what to buy, what to run, and how to keep it safe.
- **Curious practitioners** who use cloud AI services and want to understand what happens when the model runs on a machine they control.

What You Will Learn

- How language models generate text, and what tokens, context windows, parameters, and sampling mean in practice.
- How to size RAM and VRAM with simple formulas for weights and the KV cache, and how to estimate speed from memory bandwidth.
- How to choose an open model, read a model card, and check its license before you rely on it.
- What model formats and quantization (safetensors, GGUF, GPTQ, AWQ) change, and how to measure the quality cost yourself.

- How to run models with llama.cpp, Ollama, desktop applications, and Hugging Face transformers on a CPU.
- How to expose a local model through an OpenAI-compatible API, stream responses, and handle many users.
- How to build a private document assistant with local embeddings, sources, tool calling, and human review.
- How to deploy, secure, monitor, update, and govern a local AI server for a whole office.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains why and when to run models locally, how models work from an operator's point of view, hardware sizing, choosing models, and quantization. **Part II — Local Runtimes** (Chapters 6–10) covers llama.cpp, Ollama, desktop applications, transformers on the CPU, and generation and performance settings. **Part III — Building on Local Models** (Chapters 11–15) covers prompting small models, OpenAI-compatible APIs, our own FastAPI wrapper, streaming and concurrency, and local embeddings. **Part IV — Private Applications** (Chapters 16–20) builds the document assistant, tool calling, drafting and summarizing, evaluation, and a web interface. **Part V — Office Deployment** (Chapters 21–24) covers the office server, multi-user serving, security and privacy, and monitoring. **Part VI — Production and Beyond** (Chapters 25–28) covers model updates, licensing and governance, adapters and fine-tuning, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter includes hands-on examples, exercises under **Try It Yourself**, and a **Key Takeaways** box for quick review.

The Running Example: Ashcombe Legal

Throughout the book we help **Ashcombe Legal**, a fictional small law firm, build **AshDesk**, a private AI assistant. Ashcombe's AI use policy forbids sending client information to cloud chatbots, so everything must run on hardware the firm owns. Rafael Ortiz, the firm's IT and Operations Manager, starts with a small model on his laptop and ends with a shared office server that answers questions about firm policies and client documents with sources, looks up matter details through tools, drafts text for attorney review, and keeps a privacy-preserving audit log. The sample documents and matter list are small enough to read in a few minutes, so you can check every answer the assistant gives.

Versions and Conventions

The Python examples use **Python 3.12** with recent versions of PyTorch (CPU build), Hugging Face transformers, sentence-transformers, FastAPI, and uvicorn; the companion `requirements.txt` lists the exact versions they were verified with. The reference model is `Qwen/Qwen2.5-0.5B-Instruct`, a

small open model that runs on an ordinary CPU, with `sentence-transformers/all-MiniLM-L6-v2` for embeddings. Check each model's license before production use.

llama.cpp, Ollama, GPU servers, and desktop applications change quickly. Their commands are shown as documented usage for recent versions, and their output is illustrative; flag names and defaults may differ in the version you install. Timings depend entirely on your hardware, so we teach you how to measure them rather than quoting speeds you should expect. Sizes use 1 GB = 1,000,000,000 bytes unless a note says otherwise.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause data loss, errors, or security problems.

Code, commands, file names, and API fields appear in `monospace`; long lines are wrapped to fit the page. Commands for Windows PowerShell and for Linux or macOS shells are labeled when they differ. Where a chat model other than our local one could be used, we write *[your AI model]*.

Contents

Preface

PART I — FOUNDATIONS

1 Why Run LLMs Locally: Privacy, Control, Cost, and Trade-offs

2 How LLMs Work for Operators: Tokens, Context, Parameters, and Generation

3 Sizing Your Hardware: RAM, VRAM, Bandwidth, and the Memory Math

4 Choosing an Open Model: Families, Sizes, Model Cards, and Licenses

5 Model Formats and Quantization: Safetensors, GGUF, GPTQ, and AWQ

PART II — LOCAL RUNTIMES

6 llama.cpp: Building, Running, and Serving GGUF Models

7 Ollama: Pulling, Running, and Customizing Models with Modelfiles

8 Desktop Apps and GUIs: Local Chat Without the Command Line

9 Transformers on the CPU: Running Open Models from Python

10 Tuning Generation and Performance: Sampling, Context, Threads, and Speed

PART III — BUILDING ON LOCAL MODELS

11 Prompting Small Local Models: Chat Templates, System Prompts, and Structured Output

12 OpenAI-Compatible APIs: One Client, Many Local Backends

13 Building Your Own API: An OpenAI-Compatible FastAPI Wrapper

14 Streaming, Concurrency, and Queues

15 Local Embeddings and a Private Document Index

PART IV — PRIVATE APPLICATIONS

16 Private Document Q&A: Answers with Sources from Firm Documents

17 Tool Calling with Local Models: Letting the Model Ask, Letting Code Decide

18 Drafting and Summarizing Safely: Long Documents, Templates, and Human Review

19 Testing and Evaluating a Local Assistant

20 A Web Interface for Staff: A Simple, Private Chat Page

PART V — OFFICE DEPLOYMENT

21 The Office AI Server: Hardware, Operating System, and Installation Plan

22 Multi-User Serving: Authentication, Rate Limits, and Fair Queues

23 Security and Privacy: Threat Model, Network Isolation, Logging, and Retention

24 Monitoring and Operations: Health Checks, Metrics, Logs, and Capacity

PART VI — PRODUCTION AND BEYOND

25 Updating Models Safely: Versioning, Evaluation Gates, and Rollback

26 Licensing, Compliance, and Governance for Open Models

27 Adapters and Fine-Tuning: When a Local Model Needs More

28 Case Study: AshDesk Goes Live at Ashcombe Legal

APPENDICES

A Local LLM Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

Why an organization would run language models on its own hardware, what actually happens when a model generates text, how to size memory and hardware with simple arithmetic, how to choose an open model and read its license, and what quantization and model formats really change.

1 · Why Run LLMs Locally

2 · How LLMs Work for Operators

3 · Sizing Your Hardware

4 · Choosing an Open Model

5 · Model Formats and Quantization

CHAPTER 1 · PART I — FOUNDATIONS

Why Run LLMs Locally: Privacy, Control, Cost, and Trade-offs

Running a language model on hardware you own changes who can see your data, who decides when the model changes, and how you pay for it. This chapter explains what "local" really means, weighs the benefits against the honest costs, introduces Ashcombe Legal and the assistant we will build for them, and gets your Python environment ready for the rest of the book.

What "Running Locally" Means

When most people use a large language model (LLM), they type into a website or call an API run by someone else. The model, the hardware it runs on, the logs of every request, and the decision about which version you get all belong to the provider. You send text across the internet, a data center you never see processes it, and text comes back. For many tasks that arrangement is convenient and perfectly sensible.

Running an LLM *locally* means the model weights, the software that runs them, and the prompts and answers all stay on machines you control. "Local" is a spectrum rather than a single setup, and it helps to name the points on it, because each one has different costs and different guarantees:

Setup	Where the model runs	Typical user	What stays private
Laptop or desktop	Your own computer's CPU, GPU, or unified memory	One person, experiments, offline work	Everything, as long as the machine is secure
Workstation	A powerful desktop with one or more GPUs	A developer or a small team	Everything inside the office network
Office server	A dedicated machine on the local network	A whole office through an API or web page	Everything, if the network and logs are protected
Private cloud	Rented servers or GPUs you configure yourself	Larger organizations	Depends on the contract and your configuration
Cloud AI service	The provider's data centers	Anyone with an account	Whatever the provider's terms promise

This book focuses on the first three rows. We start on a laptop because that is where you learn cheaply, and we finish on an office server because that is where a team gets real value.

The thing that makes all of this possible is the rise of **open-weight models**: models whose trained parameters you can download and run yourself. Several model families are published this way, in sizes from well under one billion parameters to many tens of billions. Chapter 4 explains how to choose among them and how to read their licenses, which vary more than the word "open" suggests.

What actually travels when you use a cloud service

It is worth being precise about what leaves your building when you use a cloud AI service, because "privacy" is too vague to make decisions with. Every request carries the full prompt, which in practice means the question plus any document text, email thread, or contract clause you pasted in. The response comes back the same way. Depending on the provider and your plan, the request may be logged for abuse monitoring, kept for a period of time, reviewed by staff in some circumstances, or used to improve future models unless you opt out. Even when the terms are good, the data has crossed a boundary you do not control.

Running locally replaces that trust in a policy with control over a machine. That is the core trade: you take on responsibility for hardware, software, and security, and in exchange no prompt ever has to leave your network.

Why Organizations Run Models Locally

Teams choose local models for a handful of recurring reasons. Usually more than one applies, and it is useful to know which ones matter most to you, because they lead to different design decisions later.

Confidentiality

The most common reason is simple: some data must not leave the organization. Law firms hold privileged communications. Clinics hold health records. Engineering firms hold designs under non-disclosure agreements. For these organizations, a cloud chatbot is not just a risk to evaluate; internal policy often forbids it outright for anything that identifies a client or a person. A local model lets staff use AI on exactly the material that benefits most from it, such as long contracts and dense policies, without breaking that rule.

Data residency and control

Some organizations must know where their data is physically stored and processed, because of contracts, client requirements, or their own risk appetite. With a local model the answer is concrete: the data is on this server, in this room, and here is the access list. You also control retention. If your policy says logs are kept for 90 days, you can write the job that deletes them on day 91 and prove it ran.

Control over model versions

Cloud models change. Providers retire versions, adjust safety behavior, and update models behind a stable name. If you have tuned prompts and tested answers carefully, an unannounced change can quietly break a workflow. A local model changes only when you change it. You can pin an exact file, record its checksum, test a new version against your own questions before switching, and roll back in minutes if something goes wrong. Chapter 25 turns this into a formal update process.

Offline and air-gapped use

A local model keeps working when the internet connection is down, on a plane, on a ship, or in a secure facility with no outside network at all.

Predictable cost

Cloud AI is usually priced per token or per seat. Local AI is priced mostly up front: you buy hardware once, then pay for electricity and the time of the person who runs it. Whether that is cheaper depends entirely on how much you use it, how large a model you need, and how you value staff time. What local hosting does offer is *predictability*. Heavy use in a busy month does not create a surprise bill. We deliberately do not quote prices in this book, because hardware prices change constantly; Chapter 3 gives you the arithmetic to decide what hardware you need, and you can price it on the day you buy.

> **NOTE**
>
> Local and cloud are not mutually exclusive. Many organizations use a cloud AI service for general tasks that involve no confidential data, such as drafting a job advertisement, and a local model for everything that touches client or personal information. A clear policy that says which tool is allowed for which kind of data is often more useful than a blanket ban.

The Honest Costs

Local models are not free and they are not magic. Before you commit, look squarely at what you give up and what you take on. Every one of the following costs is real, and pretending otherwise leads to disappointed users and abandoned projects.

Hardware

A model needs memory to hold its weights, more memory to remember the conversation (the KV cache, explained in Chapter 3), and memory bandwidth to generate text at a usable speed. A small model runs on an ordinary laptop. A model good enough for serious drafting usually needs either a GPU with plenty of video memory or a machine with a large pool of fast unified memory. Serving several people at once multiplies part of that requirement. Chapter 3 shows exactly how to compute it, so you buy enough without overbuying.

Administration time

Somebody has to install the operating system and drivers, keep them patched, update the runtime and the model, watch the logs, answer "why is it slow today?", and restore the system when a disk fails. Budget for this time explicitly. A local AI server is a production system, not a gadget.

Smaller models

The largest models available through cloud services are bigger than anything most offices can run. The models you can run locally are smaller, and smaller models make more mistakes: they follow instructions less reliably, know fewer facts, and reason less well over long, complicated inputs. For focused tasks such as answering questions from a provided document, a well-chosen local model can still be very useful. But you must design for its limits: short, concrete prompts, retrieval of the right text, validation of anything the model produces, and a human who reviews the result.

No automatic improvements

A cloud model gets better without you doing anything. A local model stays exactly as good as the day you installed it until you choose to upgrade. That is the flip side of version control: stability means you must schedule and test upgrades yourself.

Security is now your job

When you run the model, you also run the security. A model server listening on all network interfaces with no password is an open door to anyone on the network. Logs that store full prompts become a second copy of your confidential data. Documents fed to the model can contain instructions that try to manipulate it. Model files from untrusted sources can, in some formats, execute code when loaded. None of these risks is exotic, and all of them have straightforward defenses, which Chapters 5, 22, and 23 cover in detail.

> **WARNING**
>
> Local is not automatically secure. Moving a model inside your network removes one risk (data going to an outside provider) and adds several of your own: exposed ports, missing authentication, logs full of client text, and unpatched software. Plan security from the first day, not after the pilot succeeds.

A Decision Checklist

Before building anything, answer the following questions honestly. They take an hour with the right people in the room and can save months of work. The more answers point to the left column, the stronger the case for running locally.

Question	Points toward local	Points toward a cloud service

Does the work involve confidential or personal data?	Yes, routinely	Rarely or never
Does policy or a contract forbid outside processing?	Yes	No
Are the tasks focused (questions over known documents, summaries, extraction)?	Mostly	Open-ended, needs the strongest model
Is someone available to run a server?	Yes, with time budgeted	Nobody has time
Do you need to control exactly when the model changes?	Yes	Not important
Will usage be steady and fairly heavy?	Yes	Occasional bursts
Must it work without an internet connection?	Yes	No

No single row decides the matter. An organization with highly confidential data but nobody to run a server may be better served by a carefully contracted private deployment than by a neglected machine in a cupboard. The checklist makes the reasons explicit, so the decision can be explained to a partner or a client who asks.

> **TIP**
>
> Write the answers to the checklist down and keep them with the project. Six months later, when someone asks why the firm did not just use a cloud service, you will have the reasoning on one page, and you will be able to check whether any of the answers have changed.

Meet Ashcombe Legal

Throughout this book we follow one organization from first experiment to production. **Ashcombe Legal** is a fictional small law firm with **18 people** in one office: 6 attorneys, 5 paralegals, and 7 operations and administrative staff. The firm handles commercial leases, employment matters, estates and probate, and small-business contracts. Like every law firm, it lives on confidential information.

Several attorneys started pasting text into cloud chatbots to summarize long documents. The managing partner stopped that quickly and wrote a policy. **ASH-POL-001, the AI Acceptable Use Policy**, is one of the six sample documents that come with this book, and its main rules shape everything we build:

- The only approved AI assistant for client work is **AshDesk**, the firm's private assistant, which runs on the firm's own server and does not send prompts or documents to any outside service. Cloud chatbots must never receive client names, matter numbers, documents, or any detail that could identify a client.
- AshDesk answers are drafts. An attorney must review any AI-assisted text before it goes to a client, a court, or another party.
- AshDesk keeps an audit log of who asked, when, and how long the answer took. The log stores a fingerprint (hash) of each prompt, not the prompt text, and entries are kept for 90 days.

- Staff report a wrong, harmful, or surprising answer to the IT and Operations Manager within one business day.

Notice what the policy does not say. It does not ban AI, and it does not demand perfection. It accepts that the assistant can be wrong, puts a human review step in place, and treats logs as sensitive.

The people

A handful of people at Ashcombe appear throughout the book. You will mostly work alongside Rafael, but every one of them has a stake in the result:

Person	Role	Interest in AshDesk
Margaret Hale	Managing Partner	Sponsor and owner of ASH-POL-001; approves models and budgets
Rafael Ortiz	IT and Operations Manager	Builds and runs AshDesk; your companion in this book
Lena Ferreira	Records and Compliance Manager	Retention rules, conflict checks, audits of the logs
Nadia Kowalski	Senior Associate, employment	Wants fast answers about contracts and policies
Samir Haddad	Associate, leases	Reviews lease summaries; an early pilot user
Ben Tran	Paralegal	Looks up matters and deadlines every day; an early pilot user

Staff email addresses follow the pattern `firstname.lastname@ashcombe.example`, using a reserved example domain. All of these people, the firm, its clients, and its matters are invented for teaching.

What AshDesk must do

Margaret, Rafael, and Lena agreed on a short list of goals for the first version. AshDesk must:

1. Answer questions about firm policies and indexed client documents in one or two sentences, and show which documents the answer came from.
2. Say plainly when it cannot find the answer, instead of guessing.
3. Look up matter details, such as the responsible attorney and the next deadline, from the firm's matter list.
4. Draft short text, such as a summary or a letter, clearly marked as a draft for attorney review.
5. Run entirely on hardware the firm owns, with no outbound connection needed during normal use.
6. Keep an audit log that records who used it and when, without storing what they asked.
7. Serve the office from 7:00 a.m. to 9:00 p.m. on weekdays, with updates on the first Saturday of each month and remote staff connecting through the firm's VPN.

Rafael also wrote down one planning number: AshDesk should handle **4 simultaneous requests** at peak. That figure is his assumption, based on how many people are likely to ask a question in the same few seconds in an 18-person office; it is not a measurement. We will use it in Chapter 3 to size memory, and Chapter 24 shows how to check it against real usage once the system is running.

The plan

Rafael's plan mirrors the structure of this book. In Parts I and II he pilots on his own **laptop, which has 16 GB of RAM and no discrete GPU**, using a small model to learn how everything works. In Parts III and IV he builds AshDesk itself: an API, document question answering with sources, tool calling against the matter list, drafting, tests, and a simple web page. In Part V he moves AshDesk to an **office AI server: one workstation-class machine with one 24 GB GPU and 64 GB of RAM, running Linux on the office network, with no inbound internet access**. Part VI covers the long life after go-live: model updates, licensing and governance, adapters, and the full case study.

We give the final production decision away now so that you can see where the numbers lead: AshDesk will run **an 8B-class instruct model, quantized to Q4_K_M in the GGUF format, served by llama.cpp's `llama-server` on 127.0.0.1:8080 behind the AshDesk API**. Ollama is evaluated as an alternative backend, and a 14B-class model is kept as the documented upgrade path. Chapters 3, 5, 6, and 21 explain every part of that sentence and why the firm chose it.

How This Book Works

We want every claim in this book to be something you can check. That leads to a few deliberate choices about tools, models, and output.

The Python sample runs on a plain CPU

The companion code is a small Python package called `ashdesk`, plus scripts and tests. It uses Hugging Face transformers with the CPU build of PyTorch, so it runs on almost any computer, including Rafael's laptop. Every Python example in the book was run, and the outputs we print come from those runs.

The model in the sample is `Qwen/Qwen2.5-0.5B-Instruct`, a small open instruct model with about half a billion parameters, released under the Apache-2.0 license. For embeddings (Chapter 15) we use `sentence-transformers/all-MiniLM-L6-v2`, also Apache-2.0. A 0.5B model is far too small for production legal work, but it is ideal for learning on a CPU. The same code serves a larger model on a GPU server; only the model name and a couple of settings change.

llama.cpp, Ollama, and GPUs: documented usage

The production runtimes, llama.cpp and Ollama, and GPU settings change quickly and depend on your hardware. For these we show commands as documented for recent versions and label their output as **illustrative**. Flag names and defaults do change between releases, so always confirm with

the tool's own `--help` or documentation for the version you install.

No borrowed speed numbers

You will not find tokens-per-second figures for specific hardware in this book. Speed depends on your exact machine, model, quantization, context length, and runtime version. Instead, Chapter 3 gives you a formula for the theoretical ceiling, and Chapter 10 gives you a script to measure your own machine. When a sample script prints a timing, treat it as a reading from your computer, not a benchmark.

> **NOTE**
> When we need to refer to a commercial cloud chat model, we write "a cloud AI service" or "[your AI model]" rather than naming a product. Open models are named only when our code really uses them. Model and runtime licenses change, so check the current license of the exact model and version you download before you rely on it.

Setting Up Your Environment

The rest of Part I is mostly concepts, with two small scripts. Setting up now means everything is ready when you need it. You need Python 3.12, a terminal, about 3 GB of free disk space for packages and the small model, and an internet connection for the one-time downloads.

Step 1: Check Python

On Windows, the Python launcher `py` picks a specific version. On Linux and macOS, use `python3` or `python3.12`, depending on how Python was installed.

Windows PowerShell

```
py -3.12 --version
```

Linux / macOS

```
python3.12 --version
```

You should see `Python 3.12.x`. Other recent Python 3 versions will probably work, but 3.12 is what the sample was verified with.

Step 2: Create a virtual environment

A virtual environment keeps the book's packages separate from everything else on your computer, so an upgrade for one project cannot break another. Create it inside the sample folder (or wherever you keep projects) and activate it.

Windows PowerShell

```
cd path\to\sample
py -3.12 -m venv .venv
.\.venv\Scripts\Activate.ps1
```

Linux / macOS

```
cd path/to/sample
python3.12 -m venv .venv
source .venv/bin/activate
```

Once the environment is active, your prompt usually shows `(.venv)`, and the command `python` refers to the environment's Python 3.12. From here on we write `python` in commands; it always means "the Python of your activated venv".

> **TIP**
>
> If PowerShell refuses to run `Activate.ps1` because script execution is disabled, you can allow scripts for your user account with `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned`. On a managed work computer, ask your IT team first; some organizations control this setting centrally for good reasons.

Step 3: Install PyTorch (CPU build)

PyTorch is published in several builds: CPU-only, and builds for different GPU platforms. The CPU build is much smaller and is all the sample needs. The PyTorch project hosts CPU wheels on its own package index; the exact install command depends on your operating system and changes over time, so check the selector on pytorch.org. At the time of writing it looks like this:

```
python -m pip install --upgrade pip
python -m pip install torch --index-url https://download.pytorch.org/whl/cpu
```

On some platforms the default package from the Python Package Index already is the CPU build; on Linux the default package typically includes GPU support and is much larger. Using the CPU index explicitly avoids downloading gigabytes of GPU libraries you will not use on a laptop. On the office server in Part V, Rafael will install a GPU-enabled runtime instead.

Step 4: Install the rest of the requirements

The sample's `requirements.txt` pins the versions the code was verified with. Pinning means the office server installs exactly what you tested.

requirements.txt

```
# AshDesk reference implementation - versions it was verified with (CPU-only PyTorch).
# Newer versions usually work; pin these for a reproducible office install.
torch==2.14.1
transformers==5.19.0
tokenizers==0.23.2
huggingface_hub==1.33.0
safetensors==0.8.0
sentence-transformers==6.1.0
numpy==2.5.3
fastapi==0.143.0
pydantic==2.13.5
uvicorn==0.54.0
httpx==0.28.1
pytest==9.1.1
```

```
python -m pip install -r requirements.txt
```

Because PyTorch is already installed from the CPU index, pip sees that the pinned version is satisfied and installs only the remaining packages.

Step 5: Download the models once

The first time a script loads `Qwen/Qwen2.5-0.5B-Instruct`, the transformers library downloads it from the Hugging Face Hub into a local cache folder. The model's weights file is 988,097,824 bytes, just under 1 GB, plus a few small configuration and tokenizer files. The embedding model is much smaller. After that first download, nothing needs the internet again.

The sample scripts deliberately turn on **offline mode** by setting the environment variables `HF_HUB_OFFLINE=1` and `TRANSFORMERS_OFFLINE=1`, so they never contact the Hub at run time. That is the behavior you want in production: a private assistant should not make surprise network calls. To allow the one-time download, run the first script with offline mode switched off for that session:

Windows PowerShell (one-time download)

```
$env:HF_HUB_OFFLINE = "0"; $env:TRANSFORMERS_OFFLINE = "0"
python scripts\01_hello_local.py
Remove-Item Env:HF_HUB_OFFLINE, Env:TRANSFORMERS_OFFLINE
```

Linux / macOS (one-time download)

```
HF_HUB_OFFLINE=0 TRANSFORMERS_OFFLINE=0 python scripts/01_hello_local.py
```

The scripts only set offline mode if the variable is not already set, which is why setting it to `0` for one run works. We look at what this first script does in Chapter 9; for now it is enough that it finishes and prints an answer. Do the same once with `03_build_index.py` in Chapter 15 to fetch the embedding model.

> **WARNING**
>
> Leave offline mode switched on for everyday runs and on the office server. With offline mode off, a library may check the Hub for newer files when it starts, and a model repository can change after you tested it. Download once, verify, pin the revision (Chapter 9 and Chapter 25), and then work offline.

Step 6: Run the tests

The sample includes nine fast tests that use a fake model instead of the real one, so they finish quickly and need no download. They check the sizing math, chunking, the tools, and the API. Running them is the quickest way to confirm your installation works.

```
python -m pytest -q tests
```

Output (timing varies)

```
.........                                                                [100%]
9 passed in N.NNs
```

"9 passed" means your environment is ready. If an import fails, check that your virtual environment is active.

A First Look at the Sample Folder

The sample is small on purpose. You can read every file in an afternoon, and every chapter that uses it points to the exact module it discusses. Here is the layout, with the chapter in which each part is explained:

```
sample/
  requirements.txt           pinned versions verified
  ashdesk/
    __init__.py              __version__ = "1.0.0"
    config.py                Settings, SYSTEM_PROMPT, NOT_FOUND
    sizing.py                memory math (Chapter 3)
    quant.py                 block quantization from scratch (Chapter 5)
    llm.py                   LocalChatModel, ChatResult (Chapter 9)
    embeddings.py            Embedder (Chapter 15)
    documents.py             Document, Chunk, chunking (Chapter 15)
    index.py                 VectorIndex, Hit (Chapter 15)
    qa.py                    answer_question (Chapter 16)
    tools.py                 TOOLS, get_matter, run_tool (Chapter 17)
    evaluate.py              golden-question evaluation (Chapter 19)
    audit.py                 privacy-preserving audit log (Chapter 23)
    server.py                OpenAI-compatible API (Chapter 13)
  docs/                      6 sample documents
  data/matters.csv           5 sample matters
  data/golden_questions.csv  10 test questions with expected answers
```