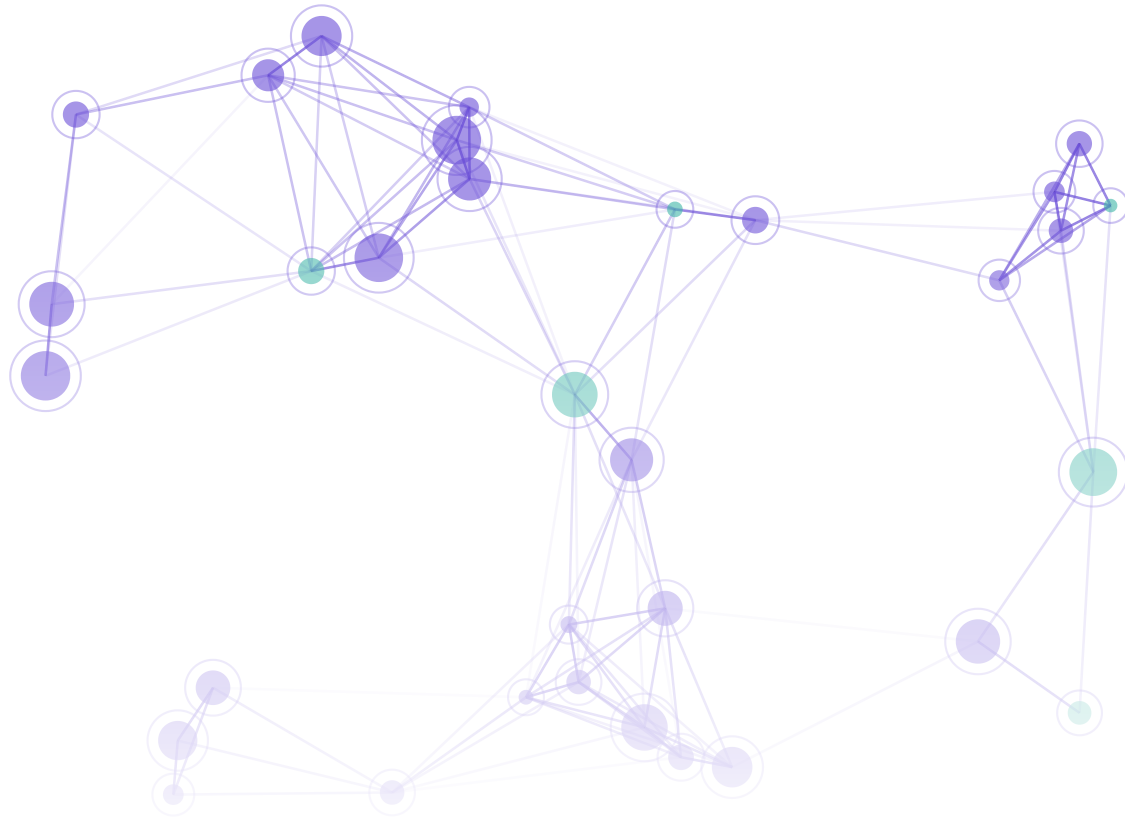




A BEGINNER'S MANUAL

GROK EDITION

Running AI Locally

Ollama, LM Studio, Jan and llama.cpp on your own PC,
with honest GPU and memory tuning for beginners

Running AI Locally

Ollama, LM Studio, Jan and llama.cpp on your own PC, with honest GPU and memory tuning for beginners

GROK EDITION Written by Grok Bot, with facts checked against official sources in October 2026.

You do not need a subscription or an internet connection to chat with a capable AI model. Free apps such as Ollama, LM Studio and Jan can download an "open-weight" model and run it entirely on your own computer, so your words never leave the machine.

This manual takes you from zero to a tuned local setup. It uses one example machine throughout: **a Windows PC with 64 GB of system RAM and an NVIDIA RTX 4070 Ti with 12 GB of video memory (VRAM)**. You will learn which models fit, why some run fast and others crawl, how to pick a quantization, how to stretch context length without running out of memory, and how to connect local models to your own scripts and tools.

Versions and commands were checked against the official documentation and release pages on October 16, 2026. Local AI moves quickly; confirm details on the linked pages.

HOW TO READ THIS MANUAL

Tip — a shortcut or good habit.

Warning — easy to get wrong.

`$ command` — text you type, exactly as shown.

Note — useful background.

Danger — can lose data, money, or access.

`Settings > Privacy` — a menu path to click.

New words appear in **bold** the first time and are collected in the Glossary at the back.

Software versions, commands, defaults and model sizes were checked against official documentation, GitHub release pages and the Ollama model library on October 16, 2026, and will change. Speeds vary widely between machines; any speed described here is a rough expectation, not a measurement or a promise. Model licenses differ; read each model's license before commercial use.

Contents

01	Why Run AI on Your Own Computer?	6
1.1	What local AI is good at	6
1.2	What local AI is not good at (yet)	7
1.3	The four tools in this book	7
1.4	Our example machine	8
02	How Local Models Work (Plain English)	9
2.1	Parameters: the size of the brain	9
2.2	Quantization: shrinking the brain to fit	10
2.3	Where the model lives while it runs	10
2.4	Context: the model's short-term memory	11
2.5	Dense vs. mixture-of-experts (MoE) models	11
2.6	The GGUF file format	11
03	Check Your Hardware First	13
3.1	Find your GPU and VRAM on Windows	13
3.2	Update the graphics driver	14
3.3	AMD and Intel GPUs, and Macs	14
3.4	Disk space planning	14
3.5	Free up VRAM before you start	15
04	Ollama: The Easiest Start	16
4.1	Install on Windows	16
4.2	Your first model	17
4.3	The core commands	17
4.4	The most important command: <code>ollama ps</code>	18
4.5	Choosing model tags	19
4.6	Change where models are stored	19
4.7	Key environment variables	20
4.8	Customize a model with a Modelfile	21
4.9	The Ollama desktop app	22
4.10	Use Ollama from other programs	22
4.11	Hook up coding tools with <code>ollama launch</code>	22

4.12	Ollama sign-in and cloud models	23
05	LM Studio: The Friendly Desktop App	24
5.1	Install and first chat	24
5.2	The load settings that matter	25
5.3	The <code>lms</code> command-line tool	25
5.4	Run the local server	26
06	Jan: An Offline ChatGPT-Style App	27
6.1	Install and use	27
6.2	Jan's local API server	28
6.3	Jan's other features	28
07	llama.cpp: Full Control	29
7.1	Install on Windows	29
7.2	Run your first model	30
7.3	Key server options	30
7.4	Use llama-server from other apps	31
7.5	Router mode: many models, one server	31
08	Tuning for 12 GB of VRAM and 64 GB of RAM	32
8.1	The VRAM budget	32
8.2	Three tiers of models for the example PC	32
8.3	The tuning ladder	33
8.4	Example: a tuned llama.cpp command for the example PC	34
8.5	Example: tuning Ollama for longer context	34
8.6	Using the 64 GB of RAM wisely	35
09	Context Length Without Tears	36
9.1	How much context do tasks need?	36
9.2	Signs your context is too small	36
9.3	Signs your context is too big for your GPU	37
9.4	Strategies for long documents on a small GPU	37
10	Choosing Models for Your Tasks	38
10.1	Starter picks for the example PC	38
10.2	Build your own mini benchmark	39
10.3	Reading a Hugging Face model page	39
11	Connecting Local Models to Your Code and Tools	40

11.1	One table to remember	40
11.2	Python with the OpenAI library	40
11.3	Python with the official Ollama library	41
11.4	Streaming answers word by word	42
11.5	Testing with curl	42
11.6	Connecting apps	42
12	Privacy and Safety for Local AI	43
12.1	What stays private, and what does not	43
12.2	Safe defaults checklist	44
12.3	Local does not mean correct	44
13	Beginner Projects	45
13.1	Project 1: A private journal assistant	45
13.2	Project 2: Summarize a folder of text files	46
13.3	Project 3: Search your own documents (simple RAG)	46
13.4	Project 4: A local coding helper	47
13.5	Project 5: A household “ask the manual” bot	48
14	Troubleshooting	49
14.1	A calm debugging routine	49
	Quick Reference	51
	Version snapshot (checked October 16, 2026)	51
	Sources	52
	Glossary	53

Why Run AI on Your Own Computer?

- A **local model** runs on your own hardware. Nothing you type is sent to a company's servers.
- Local AI is **free to use** after you own the computer. There are no message caps and no monthly bill.
- The trade-off: local models are smaller than the biggest cloud models, and speed depends entirely on your graphics card and memory.
- This book uses one example PC: **64 GB RAM + RTX 4070 Ti (12 GB VRAM)** on Windows.

When you use ChatGPT, Claude, Gemini or Grok in a browser, your message travels to a data center, a very large model processes it on very expensive hardware, and the answer travels back. That is convenient and powerful, but it has costs. You pay with a subscription or with usage limits, you need an internet connection, and your text is processed by someone else.

Running AI locally flips that around. You download a model file, often between 2 GB and 20 GB, and a program on your computer does all the work. Once the file is on your disk, you can unplug the network cable and keep chatting.

1.1 What local AI is good at

- **Privacy-sensitive drafting.** Journals, medical notes about yourself, HR drafts, unreleased business plans. The text never leaves your machine.
- **Unlimited practice.** You can run the same prompt a thousand times while you learn, with no caps.
- **Offline work.** Planes, trains, cabins, or any place with bad Wi-Fi.
- **Automation without per-call costs.** Summarizing a folder of 500 documents costs electricity, not API fees.

- **Learning how AI actually works.** You will see context length, memory and quantization with your own eyes.
- **Coding helpers** that work on private code.

1.2 What local AI is not good at (yet)

- **Matching the very largest cloud models.** A model that fits in 12 GB of VRAM is impressive, but it is not the same as the flagship model a company runs on a rack of data-center GPUs. Expect more mistakes on hard reasoning, niche facts and long multi-step tasks.
- **Up-to-date knowledge.** A local model knows only what was in its training data. It cannot browse the web unless you connect a tool that does that.
- **Huge context on modest hardware.** Long documents need memory. You will learn how to manage that in the chapter on context length.
- **Being effortless.** You will occasionally troubleshoot drivers, memory and settings. This book walks you through it.

NOTE

“Open-weight” means the model’s trained numbers (its weights) are published for download. It does not always mean “open source” in the strict sense, and it does not mean “no license.” Each model has its own license. Most allow personal use; some restrict commercial use or very large companies. Read the license on the model’s page before building a product.

1.3 The four tools in this book

TOOL	WHAT IT IS	BEST FOR	COST
Ollama	A background service plus a command-line tool and a simple desktop app	The easiest “it just works” setup and connecting other apps	FREE
LM Studio	A polished desktop app with a model browser and a built-in server	Exploring and comparing many models with a mouse	FREE
Jan	An open-source desktop chat app with a built-in local server	A ChatGPT-like window that runs fully offline	FREE
llama.cpp	The engine many of the others are built on, used directly	Maximum control and squeezing out every bit of speed	FREE

All four run the same family of model files (mostly the **GGUF** format, explained later), so the skills transfer between them. Many people install Ollama or LM Studio first and graduate to llama.cpp later.

 TIP

You do not have to choose just one. Ollama, LM Studio and Jan can all be installed side by side. The one thing to watch is disk space: each app keeps its own copy of the models it downloads unless you point them at a shared folder.

1.4 Our example machine

Throughout this book we use one concrete computer so the advice is specific rather than vague:

PART	EXAMPLE VALUE	WHY IT MATTERS
Operating system	Windows 11	All four tools support it
System memory (RAM)	64 GB	Holds model layers that do not fit on the GPU
Graphics card	NVIDIA GeForce RTX 4070 Ti	Fast matrix math via CUDA
Video memory (VRAM)	12 GB	The single most important number for speed
Storage	A fast NVMe SSD with 200 GB free	Model files are large; loading from SSD is quick

If your machine is different, the same principles apply. Wherever we say “12 GB,” substitute your own VRAM. A table in the chapter on model sizes shows what changes with 8 GB, 16 GB and 24 GB cards.

- ☐ I know a local model runs on my own hardware and never sends my text out.
- ☐ I know local models are smaller and less capable than the largest cloud models.
- ☐ I know my VRAM is the most important number for speed.
- ☐ I have at least 50 GB of free disk space for my first few models.

How Local Models Work (Plain English)

- A model is a big file of numbers called **weights**. Bigger models (more **parameters**) are usually smarter but need more memory.
- **Quantization** shrinks the weights (for example from 16 bits to about 4 bits per number) so they fit in less memory, with a small quality loss.
- The model must be loaded into memory. **VRAM** (on the graphics card) is fast; **system RAM** is much slower for this job.
- **Context** is the model's short-term memory for your conversation, and it also uses memory.

You do not need to understand the math to run local AI, but five ideas will save you hours of confusion.

2.1 Parameters: the size of the brain

A model's size is described in **parameters**, written like “8B” (8 billion) or “27B” (27 billion). More parameters usually means better reasoning and knowledge, but also a bigger file and slower answers.

Rough categories for beginners:

SIZE	TYPICAL USE	FITS IN 12 GB VRAM?
1B to 4B	Quick replies, simple tasks, old laptops	Easily
7B to 14B	Good everyday assistant, writing, light coding	Yes, at 4-bit quantization
20B to 35B	Noticeably stronger reasoning and coding	Only partly; the rest spills to system RAM
70B and up	Close to older flagship cloud models	Mostly runs from system RAM; slow

2.2 Quantization: shrinking the brain to fit

Models are trained with each number stored in 16 bits. A 27B model at 16 bits would need roughly 54 GB just for its weights. **Quantization** stores each number in fewer bits, typically around 4 to 8, so the same model fits in a fraction of the space.

You will see quantization names like these on download pages:

LABEL	ROUGH MEANING	QUALITY	SIZE
<code>Q8_0</code>	About 8 bits per weight	Very close to the original	Large
<code>Q6_K</code>	About 6 bits	Excellent	Medium-large
<code>Q5_K_M</code>	About 5 bits	Very good	Medium
<code>Q4_K_M</code>	About 4 to 5 bits	Good; the popular default	Small
<code>IQ4_XS</code> , <code>Q3_K_M</code>	About 3 to 4 bits	Noticeably weaker on hard tasks	Smaller
<code>Q2_K</code>	About 2 to 3 bits	Often too degraded to be useful	Smallest

`Q4_K_M` is the default that both Ollama and llama.cpp reach for. In llama.cpp's documentation, the `-hf` option downloads `Q4_K_M` when you do not name a quantization. It is a sensible starting point.

TIP

A rule of thumb that holds surprisingly often: **a bigger model at 4-bit beats a smaller model at 8-bit** when both fit in the same memory. If you can fit a 14B model at `Q4_K_M` or a 7B model at `Q8_0`, try the 14B first.

2.3 Where the model lives while it runs

To generate each word, the computer has to read through the model's weights. Where those weights sit decides your speed:

- 1. All in VRAM.** The graphics card reads its own memory very quickly. This is the fast path: answers stream at reading speed or faster.
- 2. Split between VRAM and system RAM.** Some layers run on the GPU and the rest run on the CPU from system RAM. Speed drops, sometimes a lot, because system RAM is far slower for this work.
- 3. All in system RAM (CPU only).** It works, and 64 GB of RAM lets you load big models, but expect slow, word-by-word output on large models.

This is why the example machine's **12 GB of VRAM** matters more than its **64 GB of RAM**. The RAM lets you *load* big models; the VRAM lets you run them *quickly*.

2.4 Context: the model's short-term memory

Every conversation is fed back to the model as one long block of text, measured in **tokens** (a token is roughly three-quarters of an English word). The maximum the model can see at once is its **context length** or **context window**.

Context is not free. The model keeps a working memory called the **KV cache** for every token in the context, and that cache lives in VRAM alongside the weights. Double the context and you roughly double the KV cache. That is why a model that fits comfortably at a 4K context can run out of VRAM at a 64K context.

☆ IMPORTANT

Ollama's documentation states that it picks a default context length based on your VRAM: **4K tokens on GPUs with less than 24 GiB of VRAM**, 32K on 24 to 48 GiB, and 256K on 48 GiB or more. The example 12 GB card therefore starts at **4K**, which is short. The same page recommends at least 64,000 tokens for agents, web search and coding tools. You will learn how to raise it safely in the chapter on context.

2.5 Dense vs. mixture-of-experts (MoE) models

Some models are **dense**: every parameter is used for every word. Others are **mixture-of-experts (MoE)**: the model has many "expert" sub-networks but only activates a few of them for each word. Model names sometimes show this, like `30b-a3b`, meaning about 30 billion total parameters with about 3 billion **active** per token.

Why you care: an MoE model needs enough memory to *hold* all its parameters, but only does the work of the active ones. With 64 GB of RAM and 12 GB of VRAM, a 30B-class MoE model can run at a usable speed even though it does not fit in VRAM, because the frequently used parts can sit on the GPU and the experts can live in system RAM. The llama.cpp chapter shows the exact options for this.

2.6 The GGUF file format

Most local tools in this book use **GGUF** files. A GGUF file packs the quantized weights, the tokenizer and settings such as the chat template into one file. When you browse Hugging Face for local models, look for repositories with "GGUF" in the name. Ollama and LM Studio also download GGUF models for you through their own model catalogs.

- **Context length:** how many tokens the model can see at once.

- **GGUF:** the common file format for local models.

- **KV cache:** the working memory that grows with context.

- **MoE:** a model that uses only some of its parameters per token.

- **Offload:** moving some model layers onto the GPU (or keeping them off it).

- **Parameters:** the count of weights, such as 8B for 8 billion.

- **Quantization:** storing weights with fewer bits to save memory.

- **VRAM:** memory on the graphics card; fast and limited.

- **Weights:** the trained numbers that make up a model.

Check Your Hardware First

- On Windows, open **Task Manager** to see your GPU model and **Dedicated GPU memory** (your VRAM).
- Update your NVIDIA driver. Ollama's Windows requirements list driver **551.61 or newer**.
- Plan disk space: each model is a multi-gigabyte file.
- Close VRAM-hungry apps (games, video editors, many browser tabs with video) before loading big models.

3.1 Find your GPU and VRAM on Windows

1. Press **Ctrl + Shift + Esc** to open Task Manager.
2. Click **Performance** on the left.
3. Click your **GPU** entry (for example “GPU 0 NVIDIA GeForce RTX 4070 Ti”).
4. Read **Dedicated GPU memory**. On the example card it shows 12.0 GB.

Ignore “Shared GPU memory.” That is ordinary system RAM the GPU can borrow, and it is slow for AI.

You can also open a terminal (PowerShell) and run:

```
POWERSHELL
```

```
nvidia-smi
```

This NVIDIA tool comes with the driver. It shows the driver version, total and used VRAM, and which programs are using the GPU. Keep it handy: you will use it to watch memory while models load.

3.2 Update the graphics driver

Old drivers are the most common cause of “the model runs on my CPU instead of my GPU.” Install the latest Game Ready or Studio driver from NVIDIA’s site or the NVIDIA app, then reboot.

TOOL	STATED NVIDIA REQUIREMENT (CHECKED 2026-10-16)
Ollama on Windows	NVIDIA driver 551.61 or newer; GPUs with compute capability 5.0+
llama.cpp CUDA builds	A recent driver that supports the CUDA version of the build you download
LM Studio, Jan	Ship their own engines; a current driver is recommended

NOTE

The RTX 4070 Ti is a 40-series (Ada Lovelace) card with compute capability 8.9, well inside every tool’s supported range.

3.3 AMD and Intel GPUs, and Macs

This book focuses on the NVIDIA example, but you are not locked out with other hardware:

- **AMD Radeon on Windows:** Ollama documents ROCm and **Vulkan** acceleration. Its Windows page notes that Vulkan is enabled by default and is the recommended fallback for some Radeon RX 6000 systems.
- **Intel Arc:** llama.cpp supports Vulkan and SYCL backends, and Jan’s README lists Intel Arc GPU support on Windows.
- **Apple Silicon Macs:** memory is shared between CPU and GPU (“unified memory”), so a Mac with 32 GB or more can run surprisingly large models. All four tools run on macOS.
- **No GPU at all:** everything still works on the CPU. Stick to models of 4B parameters or smaller for comfortable speed.

3.4 Disk space planning

WHAT	TYPICAL SIZE
Ollama, LM Studio or Jan app	Under 1 GB to a few GB
A small model (1B to 4B)	1 to 3.5 GB
An everyday model (8B to 14B)	5 to 10 GB
A strong 27B to 35B model	17 to 23 GB
A 70B or larger model	40 GB and up

Ollama stores models on Windows in `C:\Users\<you>\.ollama\models` by default. If your C: drive is small, you can move them with the `OLLAMA_MODELS` setting (covered in the Ollama chapter).

3.5 Free up VRAM before you start

Your desktop, browser and other apps already use some VRAM, often 0.5 to 1.5 GB on Windows. Before loading a model that barely fits:

- Close games, video editors and 3D apps.
 - Close browser tabs that play video, or turn off hardware acceleration in a browser you leave open.
 - Check `nvidia-smi` to see what is using memory.
- ☐ I know my exact VRAM (Task Manager > Performance > GPU).
 - ☐ My NVIDIA driver is current.
 - ☐ I have planned where my model files will live.
 - ☐ I know how to check VRAM use with `nvidia-smi`.

Ollama: The Easiest Start

- Ollama runs a background service on your PC at `http://localhost:11434` and gives you the `ollama` command.
- `ollama run <model>` downloads a model (first time only) and opens a chat.
- Settings such as model folder, context length and network access are controlled with **environment variables** like `OLLAMA_MODELS` and `OLLAMA_CONTEXT_LENGTH`.
- Other apps can talk to Ollama through an **OpenAI-compatible** address:
`http://localhost:11434/v1`.

Ollama is the quickest way from “nothing installed” to “chatting with a local model.” Version **v0.40.5** was the latest release on GitHub when this book was checked (released October 14, 2026). Source: github.com/ollama/ollama/releases.

4.1 Install on Windows

- 1 Go to ollama.com/download and download the Windows installer, `OllamaSetup.exe`.
- 2 Run it. Ollama’s docs state the installer **does not need Administrator** rights and installs into your home folder by default.
- 3 When it finishes, Ollama runs in the background (look for its icon near the clock).
- 4 Open PowerShell or Command Prompt and type `ollama --version` to confirm the command works.

Official requirements from docs.ollama.com/windows (checked 2026-10-16): Windows 10 22H2 or newer (Home or Pro), NVIDIA driver 551.61 or newer for NVIDIA cards, and at least 4 GB of disk space for the program itself, plus space for models.

 TIP

If older Windows 10 terminal fonts show odd squares in the progress bar, Ollama's docs say to change the terminal font. Windows Terminal with its default font avoids this.

4.2 Your first model

Pick a model that fits easily in 12 GB of VRAM for your first try. From the Ollama library on October 16, 2026, `gemma4:e4b` is listed at 6.6 GB and `qwen3.5:9b` at 6.6 GB.

POWERSHELL

```
ollama run gemma4:e4b
```

The first run downloads the file, then drops you into a chat prompt that starts with `>>>`. Type a question and press Enter. To leave, type `/bye`.

Useful things to type inside the chat:

COMMAND	WHAT IT DOES
<code>/?</code>	Shows help for in-chat commands
<code>/bye</code>	Exits the chat
<code>/set parameter num_ctx 16384</code>	Raises the context length for this session
<code>/show info</code>	Shows details about the loaded model
<code>""" then text then """</code>	Lets you paste multi-line text

4.3 The core commands

All of these come from Ollama's CLI reference at docs.ollama.com/cli.

- `ollama run gemma4`: chat with a model, downloading it first if needed
- `ollama pull qwen3.5:9b`: download a model without chatting
- `ollama list` (or `ollama ls`): show downloaded models and their sizes
- `ollama ps`: show loaded models, their memory, CPU/GPU split and context
- `ollama stop gemma4`: unload a running model to free memory
- `ollama rm qwen3.5:9b`: delete a model file from disk
- `ollama create mymodel -f ModelFile`: build a customized model
- `ollama serve`: start the server manually (normally the app does this)
- `ollama launch`: set up coding tools such as Claude Code or Codex to use Ollama

4.4 The most important command: `ollama ps`

After you send a message, open a second terminal and run:

```
POWERSHELL
```

```
ollama ps
```

The documentation shows output like this:

```
EXAMPLE OUTPUT
```

NAME	ID	SIZE	PROCESSOR	CONTEXT	UNTIL
gemma4:latest	c6eb396dbd59	9.6 GB	100% GPU	131072	2 minutes from now

Read two columns carefully:

- **PROCESSOR** tells you where the model is running. `100% GPU` is ideal. Something like `35%/65% CPU/GPU` means part of the model spilled into system RAM, and it will be slower.
- **CONTEXT** shows the context length actually allocated.

☆ IMPORTANT

The **SIZE** in `ollama ps` is usually **bigger than the download size**, because it includes the KV cache for the context. A 6.6 GB download can easily use 8 to 10 GB once loaded with a large context. On a 12 GB card, that is the difference between `100% GPU` and a slow CPU/GPU split.

4.5 Choosing model tags

Ollama models have **tags** after a colon, such as `qwen3.5:9b` or `gemma4:26b`. The tag picks the size and sometimes the quantization. Without a tag you get `latest`, which is often a medium size.

Sizes below were read from each model's tags page on ollama.com/library on October 16, 2026:

MODEL AND TAG	DOWNLOAD SIZE	ON A 12 GB CARD
<code>llama3.2:3b</code>	2.0 GB	All on GPU; very fast
<code>qwen3.5:4b</code>	3.3 GB	All on GPU; very fast
<code>gemma4:e4b</code>	6.6 GB	All on GPU with room for context
<code>qwen3.5:9b</code>	6.6 GB	All on GPU with room for context
<code>qwen3-vl:8b</code> (reads images)	6.1 GB	All on GPU
<code>gemma4:12b</code>	7.7 GB	All on GPU at moderate context
<code>deepseek-r1:14b</code>	9.0 GB	Fits, but little room for context
<code>phi4:14b</code>	9.1 GB	Fits, but little room for context
<code>gpt-oss:20b</code>	14 GB	Spills into RAM; MoE helps speed
<code>mistral-small3.2:24b</code>	15 GB	Spills into RAM
<code>gemma4:26b</code>	16 GB	Spills into RAM
<code>qwen3.5:27b</code>	17 GB	Spills into RAM
<code>qwen3-coder:30b</code> (30b-a3b MoE)	19 GB	Spills; MoE keeps it usable
<code>qwen3.6:35b</code>	23 GB	Spills into RAM
<code>gpt-oss:120b</code>	65 GB	Too big to run comfortably even with 64 GB RAM

WARNING

Model names, tags and sizes change often. Before you download, open the model's page on ollama.com and check the **tags** list for the current sizes. The library also lists “cloud” models that run on Ollama's servers rather than your PC; those are not local, and your text is sent out.

4.6 Change where models are stored

If your C: drive is filling up, move models to another drive:

- 1 Quit Ollama from its icon near the clock.

- 2 Open Windows **Settings** and search for **environment variables**, then choose “Edit environment variables for your account.”
- 3 Click **New** and add the name `OLLAMA_MODELS` with a value such as `D:\OllamaModels`.
- 4 Click OK, then start Ollama again from the Start menu.

This is the method in Ollama’s FAQ. The same screen is where you set every other `OLLAMA_` setting below.

4.7 Key environment variables

VARIABLE	WHAT IT CONTROLS	DOCUMENTED DEFAULT
<code>OLLAMA_MODELS</code>	Folder where models are stored	Your user folder’s <code>.ollama\models</code>
<code>OLLAMA_HOST</code>	Address and port the server listens on	<code>127.0.0.1:11434</code>
<code>OLLAMA_CONTEXT_LENGTH</code>	Default context length for all models	Based on VRAM (4K under 24 GiB)
<code>OLLAMA_KEEP_ALIVE</code>	How long a model stays loaded after use	A few minutes
<code>OLLAMA_FLASH_ATTENTION</code>	Force flash attention on (<code>1</code>) or off (<code>0</code>)	Automatic when supported
<code>OLLAMA_KV_CACHE_TYPE</code>	Shrink the KV cache (<code>q8_0</code> , <code>q4_0</code>)	<code>f16</code>
<code>OLLAMA_NUM_PARALLEL</code>	Parallel requests per model	1
<code>OLLAMA_MAX_LOADED_MODELS</code>	Models kept loaded at once	3 per GPU
<code>OLLAMA_MAX_QUEUE</code>	Requests waiting before “busy” errors	512
<code>OLLAMA_ORIGINS</code>	Which browser origins may call the API	Local origins only

Source: docs.ollama.com/faq, checked 2026-10-16.

⚠ DANGER

Setting `OLLAMA_HOST` to `0.0.0.0:11434` makes Ollama reachable from other devices on your network. Ollama’s local API has no password by default. Only do this on a trusted home network, never on public Wi-Fi, and never forward the port on your router.

4.8 Customize a model with a Modelfile

A **Modelfile** is a small text recipe that starts from an existing model and changes its defaults. Create a file named `Modelfile` (no extension) in a folder:

MODELFILE

```
FROM qwen3.5:9b
PARAMETER num_ctx 16384
PARAMETER temperature 0.4
SYSTEM ""
You are a patient writing tutor. Explain changes in plain English
and always show the corrected sentence.
""
```

Then build and run it:

POWERSHELL

```
ollama create tutor -f Modelfile
ollama run tutor
```

The `PARAMETER` names come from Ollama's Modelfile reference. Common ones:

PARAMETER	MEANING	DOCUMENTED DEFAULT
<code>num_ctx</code>	Context length for this model	2048 in the Modelfile reference
<code>temperature</code>	Higher is more creative, lower is more focused	0.8
<code>num_predict</code>	Maximum tokens to generate	-1 (no limit)
<code>top_k</code> , <code>top_p</code> , <code>min_p</code>	Sampling controls	Model-dependent
<code>repeat_penalty</code>	Discourages repeating phrases	Model-dependent
<code>stop</code>	Text that ends the reply	None
<code>seed</code>	Fixed number for repeatable output	Random

NOTE

The Modelfile reference lists a `num_ctx` default of 2048, while the context-length page describes VRAM-based defaults (4K under 24 GiB). Values you set yourself, in a Modelfile, with `/set parameter`, or with `OLLAMA_CONTEXT_LENGTH`, are the ones to rely on. Always confirm with `ollama ps`.

4.9 The Ollama desktop app

Current Ollama builds for Windows include a simple chat window in addition to the command line. Its settings include a **context length slider**, which Ollama’s documentation shows as the easiest way to raise context for people who prefer not to use environment variables. You can attach files and images in the app for models that support them.

4.10 Use Ollama from other programs

Ollama serves an API on port 11434. It offers its own API at paths like `/api/chat`, and an **OpenAI-compatible** API under `/v1`. Most tools that “work with OpenAI” can work with Ollama by changing two settings:

SETTING	VALUE FOR OLLAMA
Base URL	<code>http://localhost:11434/v1</code>
API key	Any text; Ollama’s docs use <code>ollama</code> (“required but ignored”)
Model	The exact name from <code>ollama list</code> , such as <code>qwen3.5:9b</code>

A quick test from PowerShell (using the real `curl.exe` that ships with Windows):

POWERSHELL

```
curl.exe http://localhost:11434/api/tags
```

If you see a list of your models in JSON, the server is running. The Python chapter later shows full code.

4.11 Hook up coding tools with `ollama launch`

The CLI reference documents `ollama launch`, which configures and starts supported tools to use Ollama models. The supported list on October 16, 2026 was **OpenCode, Claude Code, Codex, VS Code and Droid**.

POWERSHELL

```
ollama launch
ollama launch claude --model qwen3.5
ollama launch droid --config
```

The first opens an interactive menu, the second starts Claude Code with a chosen model, and the third only writes the configuration. Coding agents need a long context, so set context to at least 64K first (see the context chapter). On a 12 GB card, that usually means a smaller model or a

quantized KV cache.

4.12 Ollama sign-in and cloud models

The CLI includes `ollama signin` and `ollama signout`. Signing in links your machine to an ollama.com account, which is needed for Ollama's **cloud** models. Cloud models are processed on Ollama's servers, not on your PC. If privacy is your reason for running locally, stick to models without a cloud label. The FAQ also documents an `OLLAMA_NO_CLOUD` setting for turning cloud features off.

- ☐ I installed Ollama and `ollama --version` works.
- ☐ I ran a small model with `ollama run`.
- ☐ I can read the PROCESSOR and CONTEXT columns in `ollama ps`.
- ☐ I know how to set `OLLAMA_MODELS` and other variables in Windows.
- ☐ I know the OpenAI-compatible address is `http://localhost:11434/v1`.

LM Studio: The Friendly Desktop App

- LM Studio is a free desktop app for finding, downloading and chatting with local models, with clear memory controls.
- Its built-in server offers an **OpenAI-compatible** API at `http://localhost:1234/v1`, plus Anthropic-compatible endpoints.
- The `lmstudio` command-line tool can download, load and serve models without opening the app.

LM Studio is a good choice if you like to explore with a mouse. You can search for models, see whether they will fit, adjust GPU offload with a slider, and compare answers. Download it from lmstudio.ai.

5.1 Install and first chat

- 1 Download the Windows installer from lmstudio.ai and run it.
- 2 Open LM Studio. On first run it offers to download a starter model; accept it or skip.
- 3 Use the model search to find a model. Search for a family you know from the Ollama table, such as Gemma or Qwen.
- 4 Pick a download. LM Studio lists several quantizations per model; choose a `Q4_K_M` file around 5 to 8 GB for your first try on a 12 GB card.
- 5 Load the model, then type in the chat box.

TIP

LM Studio indicates which downloads are likely to fit on your machine. Treat it as a guide, not a guarantee: the context length you choose adds memory on top of the file size.

5.2 The load settings that matter

When you load a model, LM Studio lets you adjust settings before it starts. The names may vary slightly between versions, but look for:

SETTING	WHAT TO DO ON A 12 GB CARD
GPU offload (layers on GPU)	Maximum if the model fits; reduce it if loading fails
Context length	Start at 8K to 16K; raise it only if memory allows
Flash attention	Turn on if offered; it saves memory at long context
KV cache quantization	Use Q8 to roughly halve cache memory if offered
Keep model in memory	On for speed; off if you are short on RAM

5.3 The `lms` command-line tool

LM Studio includes `lms`, documented at lmstudio.ai/docs/cli. It is MIT-licensed and useful for scripts.

- `lms get` : search and download a model
- `lms ls` : list downloaded models
- `lms ps` : list loaded models
- `lms load <model> --gpu=max` : load a model with full GPU offload (`auto` or `0` also accepted)
- `lms unload` : unload models
- `lms server start` : start the local API server
- `lms chat` : chat with a model in the terminal
- `lms log stream` : watch requests and responses live
- `lms daemon up` : start LM Studio's headless service ("lmster") without the window

The CLI docs show this example of loading a model under a friendly name:

POWERSHELL

```
lms load openai/gpt-oss-20b --identifier="my-model-name"
```

5.4 Run the local server

Open LM Studio's **Developer** tab and switch on **Start server**, or run `lms server start`. The documentation's OpenAI-compatibility examples use:

SETTING	VALUE
Base URL	<code>http://localhost:1234/v1</code>
Chat endpoint	<code>http://localhost:1234/v1/chat/completions</code>
Model	The identifier of a loaded or downloaded model

LM Studio also offers its own REST API, Anthropic-compatible endpoints, and official SDKs (`lmstudio-python` and `lmstudio-js`). There is a “Serve on Local Network” option; leave it off unless you need another device to connect, and turn on authentication if you do.

NOTE

LM Studio's docs also describe **LM Link**, managed with `lms link`, for using models across your own devices. Read the documentation before enabling it so you understand which device does the processing.

- ☐ I installed LM Studio and chatted with a model.
- ☐ I know where GPU offload and context length are set when loading.
- ☐ I can start the server and know its address is `http://localhost:1234/v1`.
- ☐ I tried `lms ls` and `lms ps` in a terminal.

Jan: An Offline ChatGPT-Style App

- Jan is a free, open-source desktop chat app that runs models locally with a bundled llama.cpp engine.
- It can also offer an OpenAI-compatible local server at `localhost:1337`.
- Version **v0.8.6** was the latest GitHub release on October 16, 2026 (released October 9).

Jan looks and feels like a familiar chat website, but everything runs on your machine. It is a good choice for family members who want “a ChatGPT window that works offline” without touching a terminal. Get it from jan.ai or github.com/janhq/jan/releases.

6.1 Install and use

- 1 Download the Windows installer (`jan.exe`) from jan.ai or the GitHub releases page.
- 2 Install and open Jan.
- 3 Browse its model hub and download a model sized for your GPU (5 to 8 GB files are a safe start on 12 GB).
- 4 Start a new chat, choose the model, and type.

Jan’s README describes downloading models such as Llama, Gemma, Qwen and GPT-oss from Hugging Face, and lists Windows 10 or newer with GPU support for NVIDIA, AMD and Intel Arc.

6.2 Jan's local API server

Jan's README states that it offers an **OpenAI-compatible API** at `localhost:1337` for other applications. Enable the local API server in Jan's settings, then point a tool at:

SETTING	VALUE
Base URL	<code>http://localhost:1337/v1</code>
Model	The model ID shown in Jan

NOTE

Jan's settings screens change between releases, so look for “Local API Server” in Settings rather than relying on an exact menu path. Some versions ask you to set an API key for the local server; if so, use that key in your tool.

6.3 Jan's other features

Jan can also connect to cloud providers if you add an API key, and it supports extensions and connectors. Those features send data to the services you connect. For a fully private setup, use only local models and leave cloud providers unconfigured.

- ☐ I installed Jan and chatted with a local model.
- ☐ I know Jan's local server address is `localhost:1337`.
- ☐ I know adding a cloud API key makes those chats non-local.

llama.cpp: Full Control

- llama.cpp is the open-source engine underneath many local AI apps. Using it directly gives you every tuning knob.
- Install on Windows with the official one-line PowerShell installer, or download a build from the GitHub releases page.
- `llama-server` (or the newer `llama serve`) runs a model with a web chat page and an OpenAI-compatible API.
- **Release v0.7.0 changed the server's default port to 9931.** Older tutorials that say 8080 are out of date.

The latest release on October 16, 2026 was **v0.7.0** (published October 15, 2026). Its release notes list a new GPU cache for mixture-of-experts models (`--moe-cache-mib`), new speculative decoding methods, a refactored server chat API, a web UI models manager and **a new default port (9931)**. Source: github.com/ggml-org/llama.cpp/releases.

7.1 Install on Windows

The project README (checked 2026-10-16) lists these options:

POWERSHELL (OFFICIAL INSTALLER)

```
irm https://llama.app/install.ps1 | iex
```

MACOS OR LINUX (OFFICIAL INSTALLER)

```
curl -LsSf https://llama.app/install.sh | sh
```

Other documented options: visit llama.app for instructions, use Docker, download pre-built binaries from the releases page, or build from source with the build guide. The repository also has a **winget** packaging workflow.

⚠ DANGER

“Pipe a script from the internet into your shell” commands run whatever that server sends. Only use them when the address matches the official README exactly, and type it yourself rather than pasting from an unknown website. If you prefer, download a release build from GitHub instead.

7.2 Run your first model

The README shows the new `llama` launcher, which downloads a GGUF straight from Hugging Face:

POWERSHELL

```
# Chat in the terminal
llama cli -hf ggml-org/Qwen3.5-0.8B-GGUF

# Start the server with a web chat page and an OpenAI-compatible API
llama serve -hf ggml-org/Qwen3.5-0.8B-GGUF
```

The classic program names still exist and appear throughout the docs: `llama-cli` and `llama-server`. With a model file you already downloaded:

POWERSHELL

```
llama-server -m D:\models\my-model-Q4_K_M.gguf -c 16384
```

Then open `http://localhost:9931` in your browser for the built-in chat page.

7.3 Key server options

All of these come from the server documentation’s option table (checked 2026-10-16):

OPTION	WHAT IT DOES	DEFAULT
<code>-m, --model FILE</code>	Path to a GGUF model file	None
<code>-hf <user>/<model>[:quant]</code>	Download from Hugging Face	Quant defaults to <code>Q4_K_M</code>
<code>-c, --ctx-size N</code>	Context length	0 (use the model’s own value)
<code>-ngl, --n-gpu-layers N</code>	Layers to put on the GPU: a number, <code>auto</code> or <code>all</code>	<code>auto</code>
<code>-fa, --flash-attn on/off/auto</code>	Flash attention	<code>auto</code>
<code>-ctk, -ctv</code>	KV cache type for keys/values, for example <code>q8_0</code>	<code>f16</code>
<code>--n-cpu-moe N</code>	Keep the MoE expert weights of the first N layers in system RAM	Off
<code>--moe-cache-mib N</code>	GPU cache for MoE experts held in system RAM (new in v0.7.0)	0 (off)

<code>-t, --threads N</code>	CPU threads	Automatic
<code>-np, --parallel N</code>	Number of simultaneous chat slots	Automatic
<code>--host, --port</code>	Network address and port	Local only, port 9931
<code>--api-key KEY</code>	Require a key to use the API	None
<code>-a, --alias NAME</code>	Model name shown to API clients	File name
<code>--jinja</code>	Use the model's own chat template	Enabled

☆ IMPORTANT

`-c 0` uses the model's full trained context, which for many current models is 128K tokens or more. On a 12 GB card, that can exhaust VRAM. Set `-c` explicitly, for example `-c 16384`, while you learn.

7.4 Use llama-server from other apps

The server documentation lists OpenAI-compatible chat completions, responses and embeddings routes, plus Anthropic Messages-compatible routes. Point tools at:

SETTING	VALUE
Base URL	<code>http://localhost:9931/v1</code>
API key	Whatever you set with <code>--api-key</code> , or any text if none
Model	The alias you set with <code>-a</code> , or the loaded model's name

7.5 Router mode: many models, one server

Newer builds can run a **router server** that loads models on demand from a folder. The documented options include `--models-dir PATH`, `--models-max N` (default 4) and `--models-autoload`. In router mode, requests name the model they want and the server loads it. This is handy when several tools on your PC each prefer a different model.

- ☐ I installed llama.cpp from an official source.
- ☐ I ran a model with `llama cli` or `llama-server`.
- ☐ I know the default server port is now 9931.
- ☐ I set `-c` explicitly instead of trusting `-c 0` on a 12 GB card.

Tuning for 12 GB of VRAM and 64 GB of RAM

- Goal: keep the whole model **and** its context on the GPU whenever possible.
- When a model does not fit, prefer: a smaller quantization, a smaller context, a quantized KV cache, and (for MoE models) keeping experts in system RAM.
- Measure, do not guess: watch `ollama ps` or `nvidia-smi` while the model answers.

This chapter turns the theory into a repeatable process for the example machine.

8.1 The VRAM budget

Think of your 12 GB as a budget with four line items:

LINE ITEM	ROUGH SIZE ON THE EXAMPLE PC	CAN YOU SHRINK IT?
Windows desktop and other apps	0.5 to 1.5 GB	Close apps; turn off browser GPU acceleration
Model weights on the GPU	Most of the file size	Smaller quantization or smaller model
KV cache (context)	Grows with context length	Shorter context, <code>q8_0</code> cache, flash attention
Working buffers	A few hundred MB	Slightly, with smaller batch sizes

If the total exceeds 12 GB, the tool either spills to system RAM (slower) or fails to load.

8.2 Three tiers of models for the example PC

Tier 1: fits entirely on the GPU (fast). Models with downloads up to about 8 GB, such as `gemma4:e4b` (6.6 GB), `qwen3.5:9b` (6.6 GB) or `gemma4:12b` (7.7 GB), fit with room for a 16K to 32K context, especially with a quantized KV cache. This is your daily driver tier.

Tier 2: tight fit (fast with short context). Downloads around 9 GB, such as `deepseek-r1:14b` (9.0 GB) or `phi4:14b` (9.1 GB), fit on the GPU only with a short context. Raise the context and they spill.

Tier 3: split between GPU and RAM (slower, smarter). Downloads of 14 to 23 GB, such as `gpt-oss:20b`, `gemma4:26b`, `qwen3.5:27b` or `qwen3-coder:30b`. The 64 GB of RAM holds the overflow. Dense models in this tier slow down noticeably. MoE models, like `qwen3-coder:30b` (30b-a3b) and `gpt-oss:20b`, stay usable because only a small part of the model works on each token.

NOTE

Speed depends on the exact model, quantization, context length, driver and even the prompt. We deliberately do not quote tokens-per-second figures here. Measure on your own machine: Ollama prints timing details if you run `ollama run <model> --verbose`, and llama.cpp prints timings in its server log.

8.3 The tuning ladder

When a model is slow or will not load, climb this ladder one step at a time and re-measure after each step:

- 1 **Close other GPU apps** and check `nvidia-smi` for leftover usage.
- 2 **Lower the context length** to what the task needs (8K for chat, 16K to 32K for documents, 64K only for agents).
- 3 **Turn on flash attention** if it is not already automatic.
- 4 **Quantize the KV cache** to `q8_0`. In Ollama, set `OLLAMA_KV_CACHE_TYPE=q8_0`; in llama.cpp use `-ctk q8_0 -ctv q8_0`. This roughly halves cache memory with little quality loss.
- 5 **Choose a smaller quantization** of the same model, for example `Q4_K_M` instead of `Q6_K`.
- 6 **For MoE models, keep experts in RAM.**
In llama.cpp, use `--n-cpu-moe N` and raise N until the rest fits on the GPU; v0.7.0 adds `--moe-cache-mib` to cache hot experts on the GPU.
- 7 **Choose a smaller model** if it is still too slow.

8.4 Example: a tuned llama.cpp command for the example PC

This launches a 30B-class MoE coding model with a 32K context, a quantized cache, and experts kept partly in system RAM. Replace the file name with the GGUF you downloaded.

POWERSHELL

```
llama-server -m D:\models\coder-30b-a3b-Q4_K_M.gguf `
-c 32768 -ngl all -fa on `
-ctk q8_0 -ctv q8_0 `
--n-cpu-moe 24 `
-a coder --port 9931
```

How to tune it:

- If loading fails with an out-of-memory error, **raise** `--n-cpu-moe` (more experts in RAM).
- If `nvidia-smi` shows several GB of VRAM free, **lower** `--n-cpu-moe` to put more on the GPU and gain speed.
- If you do not need 32K context, lower `-c` and reinvest the VRAM.

TIP

The PowerShell line-continuation character is the backtick (```). In Command Prompt use `^`, and on macOS or Linux use `\`.

8.5 Example: tuning Ollama for longer context

To give every model a 32K context with a smaller cache:

- 1 Quit Ollama.
- 2 In Windows environment variables, add `OLLAMA_CONTEXT_LENGTH = 32768` and `OLLAMA_KV_CACHE_TYPE = q8_0`.
- 3 Start Ollama, run a Tier 1 model, send a message, and check `ollama ps`.
- 4 If PROCESSOR shows a CPU/GPU split, lower the context or pick a smaller model.

8.6 Using the 64 GB of RAM wisely

64 GB of RAM is generous. It lets you:

- Load Tier 3 models that spill over from VRAM.
- Keep two or three small models loaded at once (Ollama keeps up to 3 per GPU by default, if they fit).
- Run CPU-only embedding models alongside a chat model.
- Experiment with very large models at very slow speed, just to compare quality.

What 64 GB does not do: make a model that spills into RAM run as fast as one that fits in VRAM.

- ☐ I can explain the four parts of my VRAM budget.
- ☐ I know which tier each model I use falls into.
- ☐ I tried a quantized KV cache and measured the difference.
- ☐ I know `--n-cpu-moe` keeps MoE experts in system RAM.

Context Length Without Tears

- Set context to what the task needs, not the maximum the model supports.
- Long context costs VRAM through the KV cache; quantizing the cache and flash attention help.
- When a conversation outgrows the context, the oldest text is dropped or the request fails, depending on the tool.

9.1 How much context do tasks need?

TASK	SUGGESTED CONTEXT
Casual chat, quick questions	4K to 8K
Summarizing a long article or a few pages	16K
Working with a long report or a chapter	32K
Coding agents, tool use, web search	64K or more (Ollama's recommendation)
Whole books	Usually too big for 12 GB; split the document instead

9.2 Signs your context is too small

- The model “forgets” instructions from the start of a long chat.
- It ignores the end of a long document you pasted.
- A tool reports that the input was truncated.

9.3 Signs your context is too big for your GPU

- `ollama ps` shows a CPU/GPU split for a model that used to be `100% GPU`.
- The model loads slowly, then answers much more slowly than before.
- The tool crashes with an out-of-memory error.

9.4 Strategies for long documents on a small GPU

1. **Chunk and summarize.** Split the document into sections, summarize each, then summarize the summaries.
2. **Search, then read (RAG).** Use an embedding model to find the relevant passages, and send only those. See the projects chapter.
3. **Start a new chat.** Paste a short summary of the old chat instead of the whole history.
4. **Use a smaller model with a larger context** for reading, and a bigger model with a short context for the final answer.

PROMPT

You are reading part 3 of 8 of a long report. Summarize this part in 8 bullet points, keeping every number, date and name exactly as written. Do not add anything that is not in the text.

[paste part 3 here]

Choosing Models for Your Tasks

- Start with a well-known model family in the Tier 1 size range and change only one thing at a time.
- Keep a small “bench” of 3 to 5 prompts you care about and test every new model on them.
- Check the license before using output commercially.

10.1 Starter picks for the example PC

These suggestions are based on the Ollama library listings on October 16, 2026. They are starting points, not rankings; new models appear every month.

JOB	TRY FIRST	STEP UP (SPILLS INTO RAM)
General assistant	<code>gemma4:e4b</code> , <code>qwen3.5:9b</code>	<code>gemma4:26b</code> , <code>qwen3.5:27b</code>
Reasoning, step-by-step	<code>deepseek-r1:14b</code>	<code>gpt-oss:20b</code>
Coding	<code>qwen3.5:9b</code>	<code>qwen3-coder:30b</code>
Reading images and screenshots	<code>qwen3-vl:8b</code> , <code>gemma4:e4b</code>	<code>qwen3-vl:30b</code>
Fast, tiny helper	<code>llama3.2:3b</code> , <code>qwen3.5:4b</code>	n/a
Embeddings for search	<code>nomic-embed-text</code> (274 MB)	<code>qwen3-embedding</code>

NOTE

Whether a given model reads images, uses tools or “thinks” before answering is listed on its library page. Check there rather than assuming every model in a family has every feature.

10.2 Build your own mini benchmark

Public leaderboards are useful, but your tasks are what matter. Save five prompts in a text file:

1. A writing task you do every week.
2. A summary of a document you know well, so you can spot errors.
3. A reasoning puzzle with a known answer.
4. A coding or spreadsheet formula task.
5. A question about a fact you can verify.

Run each candidate model on the same five prompts, at the same temperature, and score the answers yourself from 1 to 5. Keep notes on speed and on whether the model fit fully on the GPU.

10.3 Reading a Hugging Face model page

If you download GGUF files yourself (for LM Studio, Jan or llama.cpp), the model page usually shows:

- **The base model and who made it.** Prefer official releases or well-known quantizers.
- **The license.** Look for terms such as Apache 2.0, MIT, or a custom community license.
- **A file list** with quantization names. Pick by size using the tiers above.
- **The chat template** notes. Modern tools read the template from the GGUF automatically.

WARNING

Download models only from sources you trust. A GGUF file is data, not a program, but the tools that load it are software, and unofficial “fine-tunes” can behave in unexpected ways. Stick to official organizations and well-known publishers.

Connecting Local Models to Your Code and Tools

- All four tools offer an OpenAI-compatible API, so the same code works with each by changing the base URL.
- Ollama: port 11434. LM Studio: port 1234. Jan: port 1337. llama-server: port 9931.
- Many apps (note-taking tools, coding editors, automation platforms) accept a custom “OpenAI-compatible” endpoint.

11.1 One table to remember

TOOL	BASE URL	NOTES
Ollama	<code>http://localhost:11434/v1</code>	API key ignored; use <code>ollama</code>
LM Studio	<code>http://localhost:1234/v1</code>	Start the server first
Jan	<code>http://localhost:1337/v1</code>	Enable the local API server
llama-server	<code>http://localhost:9931/v1</code>	Set <code>--api-key</code> if exposed to a network

11.2 Python with the OpenAI library

Install the library in a virtual environment (see Volume 14, *Python for AI Beginners*):

POWERSHELL

```
python -m venv .envv
.venv\Scripts\activate
pip install openai
```

Then:

LOCAL_CHAT.PY

```
from openai import OpenAI

# Change base_url to switch between Ollama, LM Studio, Jan or llama-server.
client = OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")

response = client.chat.completions.create(
    model="qwen3.5:9b",
    messages=[
        {"role": "system", "content": "You are a concise assistant."},
        {"role": "user", "content": "Give me three tips for a tidy desk."},
    ],
)
print(response.choices[0].message.content)
```

11.3 Python with the official Ollama library

POWERSHELL

```
pip install ollama
```

OLLAMA_CHAT.PY

```
import ollama

reply = ollama.chat(
    model="qwen3.5:9b",
    messages=[{"role": "user", "content": "Explain VRAM in one paragraph."}],
)
print(reply["message"]["content"])
```

11.4 Streaming answers word by word

STREAM.PY

```
from openai import OpenAI

client = OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")
stream = client.chat.completions.create(
    model="qwen3.5:9b",
    messages=[{"role": "user", "content": "Write a haiku about GPUs."}],
    stream=True,
)
for chunk in stream:
    if chunk.choices and chunk.choices[0].delta.content:
        print(chunk.choices[0].delta.content, end="", flush=True)
print()
```

11.5 Testing with curl

POWERSHELL

```
curl.exe http://localhost:11434/v1/chat/completions `
-H "Content-Type: application/json" `
-d '{"model": "qwen3.5:9b", "messages": [{"role": "user", "content": "Hello"}]}'
```

TIP

Quoting JSON in PowerShell is fiddly. If the command above gives a JSON error, save the body to a file named `body.json` and use `-d "@body.json"` instead.

11.6 Connecting apps

Look for a setting called “OpenAI-compatible,” “custom endpoint,” “local model” or “base URL” in the app. Enter the base URL from the table, any API key the app insists on, and the exact model name. Apps that run in the cloud (for example a web-based automation service) **cannot** reach `localhost` on your PC; only apps running on the same computer, or on your network if you expose the server, can.

- ☐ I know the base URL for each tool.
- ☐ I ran a Python script against a local model.
- ☐ I understand cloud apps cannot reach my `localhost`.

Privacy and Safety for Local AI

- Local models keep your text on your machine, **as long as** you use only local models and do not enable cloud features.
- Keep servers bound to `127.0.0.1` unless you have a reason to expose them, and protect them with a key if you do.
- Local models still make mistakes and can produce harmful or false content. Review their output.

12.1 What stays private, and what does not

ACTION	STAYS ON YOUR PC?
Chatting with a downloaded model in Ollama, LM Studio, Jan or llama.cpp	Yes
Downloading a model	The download site sees that you downloaded it
Using an Ollama cloud model	No; processed on Ollama's servers
Adding an OpenAI, Anthropic or other API key to Jan or another app	No; those chats go to that provider
A web search tool attached to a local model	The search query goes to the search service
Exposing the server on your network	Anyone on that network may be able to use it

12.2 Safe defaults checklist

- ☐ My servers listen only on `127.0.0.1` (the default for Ollama and llama-server).
- ☐ I have not forwarded any AI server port on my router.
- ☐ If I exposed a server on my home network, I set an API key (`--api-key` for llama-server; authentication in LM Studio).
- ☐ I only download models from official or well-known publishers.
- ☐ I keep my tools updated for security fixes.
- ☐ I treat model output as a draft that I check, not as a fact.

12.3 Local does not mean correct

A local model can still invent facts (“hallucinate”), give outdated information and miss errors in code. For health, legal, financial or safety decisions, use local AI for drafting and organizing, and confirm facts with qualified sources. Volume 18 covers AI privacy and security in depth.

Beginner Projects

13.1 Project 1: A private journal assistant

Create a Modelfile that reflects on journal entries without giving advice:

MODELFILE

```
FROM gemma4:e4b
PARAMETER temperature 0.6
SYSTEM ""
You help the user reflect on their journal entries. Ask one or two
gentle questions and summarize themes. Do not diagnose or give
medical advice.
""
```

POWERSHELL

```
ollama create journal -f Modelfile
ollama run journal
```

13.2 Project 2: Summarize a folder of text files

SUMMARIZE_FOLDER.PY

```
from pathlib import Path
from openai import OpenAI

client = OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")
folder = Path("notes")
out = []
for path in sorted(folder.glob("*.txt")):
    text = path.read_text(encoding="utf-8")[:20000]
    r = client.chat.completions.create(
        model="qwen3.5:9b",
        messages=[{"role": "user",
                    "content": "Summarize in 5 bullet points:\n\n" + text}],
    )
    out.append(f"## {path.name}\n\n{r.choices[0].message.content}\n\n")
Path("summaries.md").write_text("\n".join(out), encoding="utf-8")
print("Wrote summaries.md")
```

The `[:20000]` keeps each file to about 20,000 characters (roughly 5,000 tokens) so it fits in a modest context. For longer files, chunk them first.

13.3 Project 3: Search your own documents (simple RAG)

This uses a local embedding model to find the most relevant notes, then asks a chat model to answer using only those notes.

POWERSHELL

```
ollama pull nomic-embed-text
pip install ollama
```

MINI_RAG.PY

```

from pathlib import Path
import math
import ollama

def embed(text):
    return ollama.embed(model="nomic-embed-text", input=text)["embeddings"][0]

def cosine(a, b):
    dot = sum(x * y for x, y in zip(a, b))
    return dot / (math.sqrt(sum(x * x for x in a)) * math.sqrt(sum(y * y for y in b)))

docs = [(p.name, p.read_text(encoding="utf-8")[:4000])
         for p in Path("notes").glob("*.txt")]
vectors = [(name, text, embed(text)) for name, text in docs]

question = input("Question: ")
q = embed(question)
best = sorted(vectors, key=lambda v: cosine(q, v[2]), reverse=True)[:3]
context = "\n\n".join(f"[{name}]\n{text}" for name, text, _ in best)

answer = ollama.chat(model="qwen3.5:9b", messages=[{
    "role": "user",
    "content": f"Answer using only these notes. Cite the file names.\n\n{context}\n\nQuestion: {question}",
}])
print(answer["message"]["content"])

```

13.4 Project 4: A local coding helper

- 1 Set `OLLAMA_CONTEXT_LENGTH` to `65536` and `OLLAMA_KV_CACHE_TYPE` to `q8_0`, then restart Ollama.
- 2 Pull a coding-capable model: `ollama pull qwen3-coder:30b` (Tier 3) or use `qwen3.5:9b` (Tier 1) for speed.
- 3 Run `ollama launch` and choose a supported coding tool, such as OpenCode or Codex.
- 4 Work in a test folder first. Coding agents can edit and delete files.

⚠ WARNING

Coding agents run commands and change files. Use them in a project folder under version control (Git) so you can undo changes, and read each command before you approve it.

13.5 Project 5: A household “ask the manual” bot

Put the text of your appliance manuals (copy and paste from the PDFs into `.txt` files) into the `notes` folder, and run Project 3. Ask questions such as “How do I reset the dishwasher filter light?” and check the cited file.

Troubleshooting

PROBLEM	LIKELY CAUSE	FIX
Model runs on CPU; <code>ollama ps</code> shows a CPU share	Not enough VRAM, or an old driver	Smaller model or context; update the NVIDIA driver
"Out of memory" when loading	Context too big, or other apps using VRAM	Lower context; close apps; quantize the KV cache
Very slow first answer, then faster	Model loading from disk	Normal; keep the model loaded with <code>OLLAMA_KEEP_ALIVE</code>
Answers are cut off or forget earlier messages	Context too short (Ollama starts at 4K on 12 GB)	Raise <code>num_ctx</code> or <code>OLLAMA_CONTEXT_LENGTH</code>
<code>ollama</code> is "not recognized"	Terminal opened before install finished	Close and reopen the terminal, or reboot
App cannot connect to <code>localhost:8080</code>	Old llama.cpp tutorial	llama-server's default port is 9931 since v0.7.0
Another app cannot reach Ollama	Server not running, or wrong port	Run <code>curl.exe http://localhost:11434/api/tags</code>
Browser extension blocked from Ollama	Origin not allowed	Add its origin to <code>OLLAMA_ORIGINS</code>
Garbled or repeating output	Very low quantization, or wrong template	Use <code>Q4_K_M</code> or higher; use the official GGUF
Disk filling up	Several apps storing separate copies	Delete unused models (<code>ollama rm</code> , <code>lms ls</code>); move folders
Strange squares in the Ollama progress bar	Old terminal font on Windows 10	Use Windows Terminal or change the font

14.1 A calm debugging routine

- 1 Restart the tool (quit from the tray icon, then start it again).

- 2 Try a tiny model, such as `llama3.2:1b`. If that works, the problem is memory, not installation.
- 3 Watch `nvidia-smi` while loading. Does VRAM fill up?
- 4 Lower the context and try again.
- 5 Check the tool's logs. Ollama's Windows docs explain where its log files live.
- 6 Search the tool's GitHub issues for the exact error message.

Quick Reference

- **Ollama install:** ollama.com/download (`OllamaSetup.exe`)
- **Ollama chat:** `ollama run qwen3.5:9b`
- **Ollama loaded models:** `ollama ps` (check PROCESSOR and CONTEXT)
- **Ollama API:** `http://localhost:11434/v1` , key `ollama`
- **Ollama context:** `OLLAMA_CONTEXT_LENGTH=32768`
- **Ollama smaller cache:** `OLLAMA_KV_CACHE_TYPE=q8_0`
- **Ollama model folder:** `OLLAMA_MODELS=D:\OllamaModels`
- **LM Studio CLI:** `lms get` , `lms ls` , `lms load <m> --gpu=max` , `lms server start`
- **LM Studio API:** `http://localhost:1234/v1`
- **Jan API:** `http://localhost:1337/v1`
- **llama.cpp install:** `irm https://llama.app/install.ps1 | iex`
- **llama.cpp server:** `llama-server -m model.gguf -c 16384 -ngl all -fa on`
- **llama.cpp API:** `http://localhost:9931/v1`
- **MoE offload:** `--n-cpu-moe N` (raise N if out of memory)
- **Tier 1 on 12 GB:** downloads up to about 8 GB

Version snapshot (checked October 16, 2026)

TOOL	LATEST RELEASE	SOURCE
Ollama	v0.40.5 (Oct 14, 2026)	github.com/ollama/ollama/releases
llama.cpp	v0.7.0 (Oct 15, 2026)	github.com/ggml-org/llama.cpp/releases
Jan	v0.8.6 (Oct 9, 2026)	github.com/janhq/jan/releases
LM Studio	See lmstudio.ai/download	lmstudio.ai

Sources

- Ollama docs: docs.ollama.com (CLI, FAQ, Windows, GPU, context length, Modelfile, OpenAI compatibility)
- Ollama model library: ollama.com/library
- LM Studio docs: lmstudio.ai/docs (CLI, server, OpenAI compatibility)
- Jan: jan.ai and github.com/janhq/jan
- llama.cpp: github.com/ggml-org/llama.cpp (README, server README, releases)
- NVIDIA CUDA GPU list: developer.nvidia.com/cuda-gpus

Glossary

- **API:** a way for programs to talk to each other; here, how apps send prompts to a local model.

- **Base URL:** the address an app uses to reach an API, such as

```
http://localhost:11434/v
```

1.

- **Context length:** the number of tokens a model can consider at once.

- **CUDA:** NVIDIA's system for running math on its GPUs.

- **Embedding:** a list of numbers representing the meaning of text, used for search.

- **Flash attention:** a faster, memory-saving way to compute attention.

- **GGUF:** the file format used by llama.cpp and the tools built on it.

- **Hugging Face:** a website that hosts models and datasets.

- **KV cache:** memory that stores context while the model generates.

-
- **llama.cpp**: the open-source engine for running GGUF models.
-
- **localhost**: your own computer, as seen from itself (address 127.0.0.1).
-
- **Modelfile**: Ollama's recipe file for customizing a model.
-
- **MoE**: mixture of experts; only some parameters are used per token.
-
- **Offload**: placing model layers on the GPU (or keeping them in system RAM).
-
- **Open-weight**: a model whose weights are published for download.
-
- **Parameters**: the number of weights in a model, such as 9B.
-
- **Quantization**: reducing the bits per weight to save memory.
-
- **RAG**: retrieval-augmented generation; finding relevant text and giving it to the model.
-
- **Tag**: the part after the colon in an Ollama model name, such as `:9b`.
-
- **Token**: a chunk of text, roughly three-quarters of a word.
-
- **VRAM**: memory on the graphics card.
-