

Local & Open-Source LLMs

Run powerful language models on your own machine — no API keys, no data leaving your environment.

Last reviewed 28 Sep 2026 ·

Prerequisites: [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/)

Related: [Docker](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/) · [Fine-Tuning](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/) · [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

Challenge: Hosted LLM APIs require sending data to a third-party provider. For some workloads — internal logs, proprietary schemas, personal data — policy or regulation does not allow this.

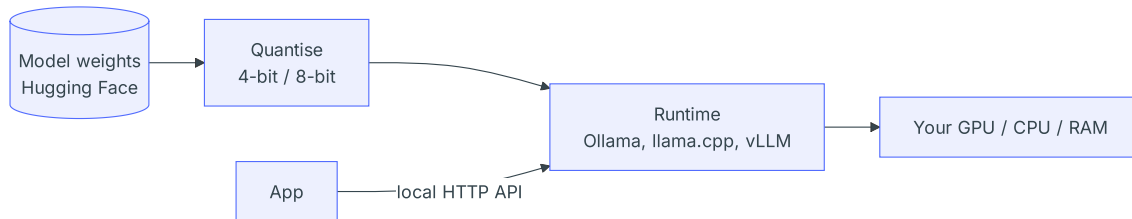
Solution: open-weight models can run entirely on a local machine or on company-controlled servers, so data never leaves the organization's infrastructure.

```
Cloud LLM:           your data → Anthropic/OpenAI API → 
response

Local LLM:           your data → model on your machine →
response
                        (nothing leaves your environment)

Trade-offs:
  Cloud:  smarter, always up to date, zero setup, pay per token
  Local:  private, free after hardware cost, needs GPU for speed,
          smaller models = slightly lower quality
```

When to use local models: - Processing data with PII (customer names, emails, medical records) - Company policy prohibits sending data to third parties - High-volume inference where cloud API costs add up - Air-gapped environments (finance, government, healthcare) - Development/testing without incurring API costs



On this page

Basic - [Ollama — Easiest Local Setup](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#ollama--easiest-local-setup) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#ollama--easiest-local-setup) - [Available Models](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#available-models) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#available-models) - [Running Models](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#running-models) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#running-models)

Intermediate - [Calling Local Models from Python](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#calling-local-models-from-python) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#calling-local-models-from-python) - [Hugging Face Transformers](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#hugging-face-transformers) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#hugging-face-transformers) - [LM Studio \(Desktop GUI\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#lm-studio-desktop-gui) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#lm-studio-desktop-gui)

Advanced - [vLLM for Production Serving](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#vllm-for-production-serving) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#vllm-for-production-serving) - [Quantization](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#quantization) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#quantization) - [Local RAG Pipeline](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#local-rag-pipeline) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#local-rag-pipeline) - [Hardware Guide](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#hardware-guide) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#hardware-guide)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/#further-reading)

Ollama — Easiest Local Setup

Ollama is the simplest way to run open-source models. One command install, one command to run.



```

# Install Ollama
# macOS
brew install ollama

# Linux
curl -fsSL https://ollama.com/install.sh | sh

# Windows – download from https://ollama.com

# Start Ollama server
ollama serve # runs on http://localhost:11434

# Pull and run a model
ollama pull llama3.2 # Meta's Llama 3.2 (3B – fast, works on CPU)
ollama pull llama3.1:8b # Llama 3.1 8B (good balance)
ollama pull mistral # Mistral 7B (good for code/SQL)
ollama pull codellama # Code Llama (specialized for code)
ollama pull deepseek-coder # DeepSeek Coder (excellent for SQL/Python)

# Chat in terminal
ollama run llama3.2
>>> Write a Python function to read a Parquet file from S3

# One-liner
ollama run mistral "What is a data lakehouse?"

```

Available Models

Open-weight models move fast. The examples use Llama 3.x, Mistral, and code models; newer families (e.g. Qwen, Gemma, newer Llama and Mistral releases, OpenAI's gpt-oss) are often better at the same size. Browse ollama.com/library (<https://ollama.com/library>) and compare on your own tasks.

Model	Size	Best for	GPU needed
Llama 3.2 3B	2GB	Quick tasks, CPU-only	No (slow on CPU)

Model	Size	Best for	GPU needed
Llama 3.1 8B	5GB	General purpose, good quality	8GB VRAM
Llama 3.1 70B	40GB	Near-GPT-4 quality	80GB VRAM (A100)
Mistral 7B	4GB	Code, SQL, reasoning	8GB VRAM
CodeLlama 7B	4GB	Code generation	8GB VRAM
DeepSeek Coder 6.7B	4GB	SQL, Python	8GB VRAM
Phi-3 Mini	2.3GB	Efficient, runs on CPU	No
Gemma2 9B	5.5GB	General, good instruction following	8GB VRAM

```
# Check what you have
ollama list

# Remove a model
ollama rm llama3.2

# Show model info
ollama show llama3.1:8b
```

Running Models

```
# Interactive chat
ollama run llama3.1:8b

# Non-interactive (pipe input)
echo "Explain what a DAG is" | ollama run llama3.1:8b

# With a system prompt (inline)
ollama run llama3.1:8b "You are a SQL expert. Write only SQL, no explanation. Query: show top 10 orders by amount"
```

```
# Create a custom modelfile (like a Dockerfile for models)
cat > Modelfile << 'EOF'
FROM llama3.1:8b
SYSTEM """
You are a data engineering expert. You answer questions
about:
SQL, Python, Spark, orchestration, streaming, and data
warehouses.
Be concise. Use code examples.
"""

PARAMETER temperature 0.1
PARAMETER num_ctx 8192
EOF

ollama create de-assistant -f Modelfile
ollama run de-assistant
```

Calling Local Models from Python

Ollama exposes an OpenAI-compatible REST API — swap the base URL and it works with the OpenAI SDK.

```
# Option 1: OpenAI SDK (compatible with Ollama)
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:11434/v1",
    api_key="ollama", # required by SDK but not validated
)

response = client.chat.completions.create(
    model="llama3.1:8b",
    messages=[
        {"role": "system", "content": "You are a SQL expert. Write only SQL."},
        {"role": "user", "content": "Count orders by status for the last 30 days"},
    ],
    temperature=0,
)
```

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

# Streaming
stream = client.chat.completions.create(
    model="llama3.1:8b",
    messages=[{"role": "user", "content": "Explain Kafka
partitions"}],
    stream=True
)
for chunk in stream:
    if chunk.choices[0].delta.content:
        print(chunk.choices[0].delta.content, end="",
flush=True)
```

```
# Option 2: Ollama Python library
# pip install ollama
import ollama

response = ollama.chat(
    model="llama3.1:8b",
    messages=[{"role": "user", "content": "What is a data
lakehouse?"}],
)
print(response["message"]["content"])

# Streaming
for chunk in ollama.chat(
    model="llama3.1:8b",
    messages=[{"role": "user", "content": "Explain window
functions"}],
    stream=True
):
    print(chunk["message"]["content"], end="", flush=True)

# Embeddings (for RAG)
response = ollama.embeddings(model="nomic-embed-text",
prompt="Apache Kafka is a streaming platform")
vec = response["embedding"] # list of floats
print(f"Dimensions: {len(vec)}") # 768 for nomic-embed-
text
```

```
# Drop-in replacement: swap cloud → local with the same
interface
import os

def get_llm_client():
    """Return the right client based on environment."""
    if os.getenv("USE_LOCAL_LLM"):
        from openai import OpenAI
        return OpenAI(base_url="http://localhost:11434/v1",
api_key="ollama"), "llama3.1:8b"
    else:
        import anthropic
        return anthropic.Anthropic(), "claude-haiku-4-5-
20251001"
```

Hugging Face Transformers

For more control — load any model from the Hugging Face Hub directly.

```
pip install transformers torch accelerate bitsandbytes
sentencepiece
```

```
from transformers import pipeline, AutoTokenizer,
AutoModelForCausalLM
import torch

MODEL = "mistralai/Mistral-7B-Instruct-v0.3"

# Quick pipeline API
pipe = pipeline(
    "text-generation",
    model=MODEL,
    torch_dtype=torch.bfloat16,
    device_map="auto",    # auto-assigns to GPU if
available
)

result = pipe(
    [{"role": "user", "content": "Write SQL to count orders
```

```

by status"}],
    max_new_tokens=256,
    do_sample=False,      # deterministic
)
print(result[0]["generated_text"][-1]["content"])

```

```

# More control with tokenizer + model directly
from transformers import AutoTokenizer,
AutoModelForCausalLM
import torch

tokenizer = AutoTokenizer.from_pretrained(MODEL)
model      = AutoModelForCausalLM.from_pretrained(
    MODEL,
    torch_dtype=torch.bfloat16,
    device_map="auto",
)

def generate(prompt: str, max_tokens: int = 256) -> str:
    messages = [{"role": "user", "content": prompt}]
    text      = tokenizer.apply_chat_template(messages,
        tokenize=False, add_generation_prompt=True)
    inputs     = tokenizer(text,
        return_tensors="pt").to(model.device)

    with torch.no_grad():
        output = model.generate(
            **inputs,
            max_new_tokens=max_tokens,
            do_sample=False,
            pad_token_id=tokenizer.eos_token_id,
        )

    new_tokens = output[0][inputs["input_ids"].shape[1]:]
    return tokenizer.decode(new_tokens,
        skip_special_tokens=True)

print(generate("Explain what a Kafka consumer group is"))

```

```

# Embeddings with sentence-transformers
# pip install sentence-transformers

```

```

from sentence_transformers import SentenceTransformer
import numpy as np

embed_model = SentenceTransformer("BAAI/bge-small-en-v1.5")
# 384 dims, 30MB

texts = [
    "Kafka is a distributed event streaming platform",
    "SQL transformations build analytics tables in the warehouse",
    "Airflow orchestrates data pipelines",
]
vecs = embed_model.encode(texts, normalize_embeddings=True)
print(vecs.shape)  # (3, 384)

# Cosine similarity (normalized vectors, so the dot product
# is the cosine similarity)
query_vec = embed_model.encode(["how to schedule a pipeline?"], normalize_embeddings=True)[0]
scores = vecs @ query_vec
print(scores)  # e.g. [0.55, 0.61, 0.56] – the Airflow doc
               # scores highest, but exact
               # numbers vary by model version; don't
               # hardcode them in an eval

```

LM Studio (Desktop GUI)

For non-technical users or quick experimentation — a desktop app with a ChatGPT-like interface.

1. Download from <https://lmstudio.ai>
2. Search for a model (e.g., "Meta Llama 3.1 8B Instruct")
3. Click Download
4. Load the model
5. Chat in the UI – or enable the local API server on port 1234

```

# LM Studio also exposes an OpenAI-compatible API
from openai import OpenAI

```

```

client = OpenAI(base_url="http://localhost:1234/v1",
api_key="lm-studio")
response = client.chat.completions.create(
    model="meta-llama-3.1-8b-instruct",    # model loaded in
LM Studio
    messages=[{"role": "user", "content": "What is Delta
Lake?"}]
)
print(response.choices[0].message.content)

```

vLLM for Production Serving

vLLM is the fastest open-source LLM serving framework — 2-24x higher throughput than a naive setup.

```

pip install vllm

# Start the server
vllm serve meta-llama/Meta-Llama-3.1-8B-Instruct \
    --port 8000 \
    --tensor-parallel-size 1 \           # use 1 GPU
    --max-model-len 8192

# Or with quantization (less GPU memory)
vllm serve meta-llama/Meta-Llama-3.1-8B-Instruct \
    --quantization awq \                # 4-bit quantization
    --max-model-len 8192

```

```

# Use exactly like OpenAI API
from openai import OpenAI

client = OpenAI(base_url="http://localhost:8000/v1",
api_key="vllm")

# Batch 100 requests simultaneously – vLLM handles
parallelism automatically
import asyncio
from openai import AsyncOpenAI

async_client =

```

```

AsyncOpenAI(base_url="http://localhost:8000/v1",
api_key="vllm")

async def classify(text: str) -> str:
    resp = await async_client.chat.completions.create(
        model="meta-llama/Meta-Llama-3.1-8B-Instruct",
        messages=[{"role": "user", "content": f"Classify as
ERROR/WARN/INFO: {text}"}],
        max_tokens=10,
        temperature=0,
    )
    return resp.choices[0].message.content.strip()

async def classify_all(logs: list[str]) -> list[str]:
    return await asyncio.gather(*[classify(log) for log in
logs])

# 100 concurrent requests
results = asyncio.run(classify_all(log_messages[:100]))

```

Quantization

Run larger models on less GPU memory by compressing weights.

```

Full precision (fp32):  7B model = 28GB VRAM
Half precision (fp16): 7B model = 14GB VRAM
8-bit quantization:    7B model = 7GB VRAM  (small quality
loss)
4-bit quantization:    7B model = 4GB VRAM  (minor quality
loss for most tasks)

```

Rule: 4-bit is usually fine for text classification, extraction, SQL generation.
 Prefer fp16 for nuanced reasoning or generation tasks.

```

# 4-bit quantization with bitsandbytes
from transformers import AutoModelForCausalLM,
BitsAndBytesConfig

```

```
import torch

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.bfloat16,
)

model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Meta-Llama-3.1-8B-Instruct",
    quantization_config=bnb_config,
    device_map="auto",
)
# 8B model now fits in ~5GB VRAM
```

Local RAG Pipeline

Full RAG pipeline with no external API calls.

```
from sentence_transformers import SentenceTransformer
import numpy as np
import llama

# — Embed with local model

embed_model = SentenceTransformer("BAAI/bge-small-en-v1.5")

documents = [
    "The orders table has: order_id, customer_id, amount, status, created_at",
    "Airflow DAGs are defined in Python. Each DAG has tasks and dependencies.",
    "Kafka topics are split into partitions for parallel consumption.",
]

doc_vecs = embed_model.encode(documents,
                               normalize_embeddings=True)
```

— Retrieve

```
def retrieve(query: str, k: int = 3) -> list[dict]:
    q_vec = embed_model.encode([query],
normalize_embeddings=True)[0]
    scores = doc_vecs @ q_vec
    top_k = np.argsort(scores)[::-1][:k]
    return [{"text": documents[i], "score":
float(scores[i])} for i in top_k]
```

— Generate with local Llama

```
def rag_answer(question: str) -> str:
    chunks = retrieve(question)
    context = "\n".join(f"[{i+1}] {c['text']}" for i, c in
enumerate(chunks))

    response = ollama.chat(
        model="llama3.1:8b",
        messages=[
            {
                "role": "system",
                "content": "Answer using ONLY the context.
If unsure, say so. Cite [1], [2] etc."
            },
            {
                "role": "user",
                "content":
f"Context:\n{context}\n\nQuestion: {question}"
            }
        ],
        options={"temperature": 0}
    )
    return response["message"]["content"]

print(rag_answer("What columns does the orders table
have?"))
# Runs entirely locally – zero API calls
```

Hardware Guide

CPU-only (no GPU):

Works but slow (~2-10 tokens/sec)

Usable for: batch jobs, testing, low-volume use

Best models: Phi-3 Mini (2.3GB), Llama 3.2 3B (2GB)

Consumer GPU (RTX 3090/4090 – 24GB VRAM):

Fast (~30-80 tokens/sec)

Fits: 7-8B models at fp16, up to ~30B models at 4-bit

(70B at 4-bit needs ~40GB – two 24GB cards or one 48GB card)

Best models: Llama 3.1 8B, Mistral 7B, CodeLlama 13B (4-bit)

Data center GPU (A10G – 24GB, A100/H100 – 80GB):

Very fast (~100-200 tokens/sec)

Fits on one 80GB GPU: 70B at 4-bit or 8-bit (~70GB); fp16 70B (~140GB) needs 2+ GPUs

Best models: Llama 3.1 70B, Mixtral 8x7B

Apple Silicon (M1/M2/M3 – unified memory):

Good performance, CPU+GPU share memory

M1 Pro (16GB): 7B models fast, 13B OK

M2 Max (96GB): can run 70B models

Use: Ollama on Mac – "just works"

Memory requirement guide (weights only – add 10-30% for KV cache and runtime,

more for long contexts or many concurrent requests):

Model size (billion params) × 2 = GB for fp16/bf16

Model size × 1 = GB for 8-bit

Model size × ~0.55 = GB for 4-bit (e.g.

Q4_K_M)

Example: 7B model = 14GB fp16, ~7GB 8-bit, ~4GB 4-bit

Common Pitfalls

Pitfall	Symptom	Fix
Expecting frontier-model quality from a small local model	Wrong SQL, missed instructions, weak reasoning	Evaluate on your own tasks; use local models for narrow, well-defined work, or fine-tune
Sizing hardware from parameter count alone	Out-of-memory errors under real load	Budget for weights <i>plus</i> KV cache (context length × concurrency)
Default context window (e.g. 2–4k tokens in Ollama)	Long prompts silently truncated; RAG answers ignore the context	Raise <code>num_ctx</code> (Ollama) or <code>--max-model-len</code> (vLLM), within memory limits
Over-aggressive quantization (2–3 bit)	Noticeably worse output	Start at 4-bit (Q4_K_M) or 8-bit; compare quality on your eval set
Using Ollama for high-concurrency production serving	Low throughput, long queues	vLLM, SGLang, or TGI with continuous batching
Mismatched chat template	Rambling output or ignored instructions	Use the model's own chat template (the tools apply it; watch custom setups)
Assuming "local" means "safe" by default	An exposed endpoint on the network with no auth	Bind to localhost or put it behind auth; keep Ollama/vLLM ports off the public internet
Model licences ignored	Legal risk in commercial use	Check each model's licence (Llama, Gemma, Qwen, Mistral all differ)
CPU-only inference for batch jobs at scale	Jobs take days	A GPU (even a cloud spot instance) or a hosted API for large batches

Cheat Sheet

Task	Command
Install Ollama (Linux)	<code>curl -fsSL https://ollama.com/install.sh sh</code>

Task	Command
Download / run a model	<code>ollama pull llama3.1:8b</code> · <code>ollama run llama3.1:8b</code>
List / remove / inspect	<code>ollama list</code> · <code>ollama rm <model></code> · <code>ollama show <model></code>
What's loaded in memory	<code>ollama ps</code>
Custom model with a system prompt	<code>Modelfile</code> with <code>FROM</code> + <code>SYSTEM</code> + <code>PARAMETER num_ctx 8192</code> → <code>ollama create my-model -f Modelfile</code>
OpenAI-compatible endpoint	Ollama <code>http://localhost:11434/v1</code> · vLLM <code>http://localhost:8000/v1</code>
Python client	<code>OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")</code>
Local embeddings	<code>ollama pull nomic-embed-text</code>
Production serving	<code>vllm serve <hf-model-id> --max-model-len 8192 --gpu-memory-utilization 0.9</code>
Check the GPU	<code>nvidia-smi</code> (NVIDIA) · Activity Monitor → GPU (Apple Silicon)

Which tool? Trying models on a laptop → Ollama or LM Studio · Python experiments and fine-tuning → Hugging Face Transformers · serving many concurrent users → vLLM / SGLang / TGI · Apple Silicon → Ollama or MLX

Quantization formats: GGUF (llama.cpp, Ollama, LM Studio — CPU/Apple/GPU) · AWQ / GPTQ (GPU serving with vLLM) · bitsandbytes 4/8-bit (Transformers, QLoRA training)

Local vs API: choose local when data can't leave your network, for offline use, or at high, steady volume · choose an API for the best quality, spiky workloads, or when you don't want to run GPUs

Interview Questions

Q: Why would a data team choose a local LLM over Claude or GPT-4? A: Three main reasons: (1) Data privacy — customer PII or proprietary schemas can't leave the environment; (2) Cost — at high volume (millions of calls/day), a self-hosted 8B model on a \$3/hour GPU instance is far cheaper than cloud API costs; (3) Latency — no network round-trip, and inference can run in parallel with data processing.

Q: What is quantization and when would you use 4-bit vs fp16? A: Quantization reduces the number of bits used to represent each model weight, reducing memory requirements. 4-bit quantization (e.g., AWQ, GPTQ) fits a 7B model in ~4GB VRAM vs ~14GB for fp16, with typically <5% quality loss on structured tasks like classification or SQL generation. Use fp16 for nuanced reasoning or creative generation where quality matters more. Use 4-bit when fitting the model on available hardware is the constraint.

Q: What is Ollama and how does it differ from vLLM? A: Both serve local LLMs, but for different use cases. Ollama is a developer-friendly tool for running models locally with a simple CLI and OpenAI-compatible API — great for development and single-user inference. vLLM is a production inference server focused on maximum throughput via PagedAttention and continuous batching — designed for serving hundreds of concurrent requests, 2-24x faster than naive serving. Use Ollama for development; vLLM for production deployment.

Further Reading

- [Ollama documentation](https://github.com/ollama/ollama/tree/main/docs) (https://github.com/ollama/ollama/tree/main/docs) and [model library](https://ollama.com/library) (https://ollama.com/library)
 - [vLLM documentation](https://docs.vllm.ai/) (https://docs.vllm.ai/)
 - [llama.cpp](https://github.com/ggml-org/llama.cpp) (https://github.com/ggml-org/llama.cpp) — the engine behind GGUF and many local tools
 - [Hugging Face Open LLM Leaderboard](https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard) (https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard)
 - [LM Studio](https://lmstudio.ai/docs) (https://lmstudio.ai/docs)
 - [Fine-Tuning LLMs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/) — adapting an open model to your task
-

Previous: [AI Observability](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026