Local AI Models Course

Flavio Copes

Updated August 11, 2026

Contents

Understand open-weight models, choose one that fits your hardware, run it locally, and build a reliable private AI feature with Ollama and Node.js.

What you'll learn: Choose and run an open-weight model locally, connect it to a small Node.js application, constrain and test its output, and document the privacy, reliability, and operating boundaries.

Understand open weights

Separate weights, architecture, training, inference, licenses, and the wider meaning of open source AI.

Weights are learned numbers

Understand model weights as the learned numerical state produced during training rather than source code or stored answers.

A language model contains a huge collection of numbers called **parameters**. Most of those parameters are **weights**.

Think of the model as an enormous audio mixing desk. Its architecture defines how the desk is wired. The weights are the positions of billions of small knobs.

Before training, those values are mostly random. The model cannot produce useful language yet.

During training, the model receives examples and tries to predict what comes next. Each error causes tiny adjustments to the weights. Repeating that process across a large dataset gradually produces a useful model.

The final weights do not contain a list of complete answers. They encode patterns that influence how an input is transformed and which token becomes likely next.

This is why a model can combine ideas in a way that does not match one training document. It is also why the model can invent a plausible answer. The weights produce likely continuations, not a database lookup with guaranteed facts.

When a model is called `1B`, it has roughly one billion parameters. A `30B` model has roughly 30 billion. Parameter count affects memory and compute, but it does not tell you whether the model is good at your task.

> The weights are the model's learned state. They are the result of training, not the training process itself.

Separate training from inference

Follow a model through training, checkpoint creation, and inference so each stage has a clear purpose and cost.

Training is the learning phase. **Inference** is the using phase.

During training, software reads batches of data, asks the model to predict an answer, measures the error, and updates the weights. Training also keeps temporary state such as gradients, optimizer

values, and the current position in the data.

A training system saves **checkpoints** along the way. A checkpoint lets the run resume or gives researchers two stages they can compare.

After training, the final weights can be copied into a smaller inference package. Inference does not update the weights for every prompt. It loads them and uses them to calculate an output.

The path looks like this:

```
training data -> training code -> checkpoints -> released weights
                                                        |
                                                        v
                                                  inference runtime
                                                        |
                                                        v
                                                    response
```

Downloading released weights does not give you the original dataset, optimizer state, or exact training environment. It gives you enough learned state to run the model, and often enough to fine-tune it further.

Keep this distinction in mind when someone says a model is open. We need to ask *which part* is open.

Architecture, weights, and runtime

Separate the model design, learned parameters, tokenizer, and inference software that cooperate during a local response.

A local AI application has several parts. Calling all of them "the model" hides useful differences.

The **architecture** describes the model shape: its layers, attention mechanism, dimensions, and how values move through the network.

The **weights** fill that architecture with learned numbers.

The **tokenizer** converts text into token IDs the model understands, then converts generated IDs back into text.

The **runtime** loads these files and performs the calculations. Ollama, llama.cpp, MLX LM, and Transformers are examples of runtimes or runtime toolkits.

A model also needs configuration and a chat template. The template turns conversation roles into the exact token pattern used during instruction training. A wrong template can make good weights behave badly.

The complete path is:

```
prompt -> chat template -> tokenizer -> architecture + weights -> token IDs -> text
```

This explains why one model file does not run inside every application. The runtime must support the architecture, file format, tokenizer, and features you need.

Open weights and open source AI

Use precise language for downloadable model parameters and understand why open weights alone do not reproduce an AI system.

An **open-weight model** makes its learned parameters available to download under some set of terms.

This can let you run the model locally, keep a fixed version, study it, quantize it, fine-tune it, or redistribute it. The license decides which actions are allowed.

Open weights are not automatically the same as **open source AI**.

With ordinary software, source code is the preferred form for studying and modifying a program. A compiled binary may be useful, but it does not show everything used to produce it.

Weights have a similar limit. You can run and modify them, but they may not reveal the training data, how the data was filtered, or the complete training code and settings.

The Open Source Initiative definition asks for the freedoms to use, study, modify, and share the system. It also describes the code, parameters, and detailed data information needed to exercise those freedoms.

This does not make an open-weight release unimportant. Downloadable weights can provide enormous practical freedom. It only means we should describe what was released accurately.

My advice is to say **open weight** when you know the parameters are available, then inspect the rest of the release before calling the complete system open source.

Read the model card and license

Inspect intended use, limitations, artifacts, provenance, and legal terms before downloading or building on a model.

A model page is not just a download button.

Start with the **model card**. It should explain what the model does, its intended uses, known limitations, training or fine-tuning information, datasets, and evaluation results.

Then read the license itself. A familiar label such as Apache 2.0 is different from a custom model license with restrictions. Do not assume that "free to download" means "free to use commercially" or "free to redistribute."

Check these fields:

- who published the repository
- the exact base model and version
- license and additional usage terms
- supported languages and tasks
- architecture and runtime requirements
- available full-precision and quantized artifacts
- context length and prompt template
- evaluations and their conditions
- limitations and out-of-scope uses

Also inspect the files. A community quantization is a new artifact maintained by someone other than the original model creator. Check its source model, conversion notes, hashes when supplied, and community history.

A detailed model card does not prove every claim. It gives you a starting point for your own evaluation.

Check your understanding: open weights

Check weights, training, inference, architecture, runtimes, model cards, licenses, and the open-source distinction.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Choose a model that fits

Compare capabilities, parameter counts, quantization, formats, context, memory, and task quality.

Start from the task

Choose a local model from a concrete workload and acceptance test rather than a leaderboard or parameter count alone.

Do not start by asking, "What is the best local model?"

Start with the task.

A model that summarizes short private notes needs different strengths from one that reads screenshots, writes code, or calls tools across a long workflow.

Write one sentence describing the input and output:

```
Given one daily activity record, return one factual sentence under 25 words.
```

Now collect representative examples. Include normal cases, empty input, unusual wording, long input, and content the model must not invent.

Define what good means. For the sentence above, we can check factual consistency, length, tone, latency, and whether the model adds unsupported details.

Only then compare models. Start with the smallest model that plausibly supports the task. A smaller model is easier to fit, faster to load, and cheaper to operate.

A leaderboard can help you create a shortlist. Your own examples decide which model belongs in the application.

Parameters and memory

Estimate the memory required by model weights and leave room for context, runtime buffers, and the operating system.

The weight estimate starts with one formula:

```
weight GB = parameters in billions x bits per weight / 8
```

An 8-billion-parameter model stored at 16 bits needs about 16 GB for its weights:

```
8 x 16 / 8 = 16 GB
```

At roughly 4 bits, the same estimate becomes 4 GB:

```
8 x 4 / 8 = 4 GB
```

Real files include scales and metadata. At runtime you also need memory for the context cache, compute buffers, the runtime, the operating system, and other applications.

On Apple Silicon, the CPU and GPU share unified memory. That makes large local models practical, but macOS still needs part of the same pool. On a discrete GPU, model layers can sometimes spill into system RAM, but generation usually becomes slower.

Treat "it fits" and "it runs comfortably" as different claims. Leave headroom, then measure on the real machine.

Understand quantization

Use lower-precision weights to reduce memory while treating quality and runtime compatibility as measured tradeoffs.

Quantization represents weights with fewer bits.

A full-precision release might use BF16 or FP16. Local variants commonly use 8, 6, 5, 4, or fewer bits per weight.

Reducing precision makes the file smaller and usually reduces memory use. It may also improve speed when the hardware and runtime have efficient kernels for that format.

The tradeoff is quantization error. Two nearby original values may become the same lower-precision value. Good methods choose scales and blocks carefully so useful behavior survives.

Names such as `Q4_K_M` describe a particular GGUF quantization recipe. The number is not a complete quality score. A strong 4-bit quantization can be a better practical choice than a poorly supported higher-precision file.

My advice is to start around 4 bits for a local experiment. Move upward when you have enough memory and your evaluation shows a meaningful improvement. Move downward only when fitting the model matters more than the quality you lose.

Always compare the exact artifacts. "The 8B model" is incomplete when one test used BF16 and another used a 4-bit conversion.

Safetensors and GGUF

Recognize two common model formats and choose between training-oriented tensor storage and portable local inference packages.

`safetensors` and `GGUF` solve related but different problems.

Safetensors stores tensors as data without using Python pickle. It is designed for safe and fast loading, including partial and zero-copy access. Model repositories often split large weights across several `.safetensors` files.

GGUF is a single-file format designed for GGML-based inference runtimes. It stores tensors plus standardized metadata. It also supports many quantization types.

You will often see the same model published both ways:

```
model-00001-of-00004.safetensors
model-00002-of-00004.safetensors
...
model-Q4_K_M.gguf
```

Use the format your runtime expects. llama.cpp loads GGUF. Ollama can manage compatible models for you. Transformers commonly loads Safetensors repositories. MLX models use artifacts prepared for Apple's MLX ecosystem.

A conversion is another build step. Record the original model, converter version, quantization method, output hash, and test results when reproducibility matters.

Context and the KV cache

Understand why long prompts consume additional memory and why advertised context length is not a free operating target.

The weights are not the only large allocation.

While generating, a transformer stores intermediate attention values for tokens it has already processed. This is the **KV cache**.

The cache prevents the model from recalculating the complete conversation before every new token. It also grows as the context grows.

The exact size depends on the number of layers, key-value heads, head dimension, cache precision, batch size, and token count. Two models with the same parameter count can have different cache requirements.

An advertised 128K context means the architecture and runtime can support that window under some conditions. It does not mean 128K will be fast, accurate, or memory-efficient on your laptop.

Begin with the shortest context that contains the information needed for the task. Measure memory and latency at the real input sizes, including the output you reserve.

Long context is useful when the model needs it. It is not a replacement for selecting relevant input.

Build a small evaluation

Compare candidate models with stable examples, explicit checks, recorded settings, and failures you can inspect.

A useful evaluation can start with 20 examples.

Save each input, the expected properties, and any reference output that helps. Include cases where the correct behavior is to say there is not enough information.

For a summarizer, checks might include:

- every named fact came from the input
- output stays under the requested length
- empty input produces no invented activity
- the JSON shape validates
- latency stays inside the product budget

Run every candidate with the same prompt, context, quantization, generation settings, and hardware. Record the exact model tag or file hash.

Some checks can be automatic. JSON can validate against a schema. Length can be counted. Required facts can be compared.

Quality judgments may still need a person. Keep the rubric short enough that two runs can be compared consistently.

When a model fails, save the case. Your evaluation should grow from real failures, not only from examples the model already handles.

Check your understanding: choosing a model

Check task selection, model size, memory, quantization, formats, context, and repeatable evaluations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Run models locally

Install Ollama, manage model files, use the local API, stream responses, and measure performance.

Choose a local runtime

Compare Ollama, llama.cpp, MLX LM, and desktop interfaces by the job they perform rather than treating them as model families.

A runtime is the software that loads a model and performs inference.

Ollama manages model downloads, local serving, and a simple API. It is a good starting point for this course.

llama.cpp is a lightweight C and C++ inference project with broad hardware support. It runs GGUF models and includes an HTTP server with compatible API routes.

MLX LM targets Apple Silicon. It can generate, quantize, and fine-tune models using Apple's MLX framework.

Desktop applications such as LM Studio add model discovery and a graphical chat interface. They can be convenient, but the model, format, and runtime compatibility still matter underneath the UI.

Choose based on the boundary you need:

- quick local chat and API: Ollama
- direct GGUF control and broad backends: llama.cpp
- Apple Silicon experiments and fine-tuning: MLX LM
- graphical exploration: a desktop interface

The model is portable only when another runtime supports its architecture, format, template, and features.

Install and check Ollama

Install Ollama, verify the local service, and distinguish a missing runtime from a missing model.

Download Ollama for macOS, Windows, or Linux from the official site and install it.

Open a terminal and check the installed version:

```
ollama --version
```

Then ask the local API which models are available:

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

A JSON response means the service is reachable. An empty model list is fine. It means the runtime works but you have not pulled a model yet.

A connection error means the service is not listening. Start the Ollama application or service before changing application code.

This small health check gives us two separate facts:

```
runtime reachable?
requested model installed?
```

Keep those checks separate in a real application. "AI unavailable" is harder to diagnose than "Ollama is not running" or "gemma3:1b is not installed."

Pull and run your first model

Download a small quantized model, start an interactive session, and confirm which artifact is stored locally.

Start with a small model so the first experiment works on modest hardware.

This course uses Gemma 3 1B:

```
ollama run gemma3:1b
```

The first run downloads an approximately 815 MB Q4_K_M artifact. Later runs use the local copy.

At the prompt, enter:

```
Explain what a model weight is in two sentences.
```

Leave the chat with:

```
/bye
```

List stored models:

```
ollama list
```

The 1B model is useful for checking the complete pipeline. Do not use its successful installation as evidence that it can handle every application task. We will evaluate quality separately.

If your machine has more memory, you can later compare a larger model using the same evaluation examples.

Call the local chat API

Send a complete chat request with curl and inspect content, token counts, loading time, and generation time in the response.

Ollama serves an HTTP API on `localhost:11434` by default.

Send one non-streaming chat request:

```
curl http://localhost:11434/api/chat -d '{
  "model": "gemma3:1b",
```

```
  "messages": [
    {"role": "user", "content": "Explain local inference in one sentence."}
  ],
  "stream": false
}'
```

The generated text is inside `message.content`.

The final response also includes measurements such as `load_duration`, `prompt_eval_count`, `prompt_eval_duration`, `eval_count`, and `eval_duration`.

Durations use nanoseconds. Generated tokens per second can be estimated with:

```
eval_count / (eval_duration / 1,000,000,000)
```

The first request may include model loading. A later request can be faster while the model remains in memory.

Do not expose the Ollama port directly to an untrusted network. A local service normally has no reason to accept requests from every device or website.

Call Ollama from Node.js

Use the built-in fetch API to connect a Node.js program to the local model without adding an SDK.

Node includes `fetch()`, so the simplest client needs no dependency.

Create `ask.js`:

```
const response = await fetch('http://localhost:11434/api/chat', {
  method: 'POST',
  headers: { 'content-type': 'application/json' },
  body: JSON.stringify({
    model: 'gemma3:1b',
    messages: [
      { role: 'user', content: 'Explain local inference in one sentence.' },
    ],
    stream: false,
  }),
})

if (!response.ok) {
  throw new Error(`Ollama returned ${response.status}`)
}

const data = await response.json()
console.log(data.message.content)
```

Run it:

```
node ask.js
```

Check `response.ok` before parsing a success payload. A local service can be unavailable, return a malformed request error, or reject an unknown model.

Keep the URL and model name in configuration when the project grows. Do not scatter them

across every feature.

Stream and measure responses

Read newline-delimited streaming events, measure time to first token, and separate model loading from generation speed.

Streaming lets the interface show text before the complete response is ready.

Ollama streams newline-delimited JSON by default. Each line is a complete JSON object, not one fragment of a larger JSON document.

The reading loop has four steps:

```
read bytes -> decode text -> split complete lines -> parse each JSON object
```

Keep any unfinished final line for the next chunk. Network chunks do not have to end at JSON boundaries.

Measure at least three timings:

- request start to first generated content
- request start to final event
- generated tokens per second from the final counters

Run a cold request after the model has been unloaded, then a warm request. This separates loading time from prompt processing and generation.

For a background summarizer, total latency may matter most. For an interactive chat, time to first token changes how fast the application feels.

Performance numbers need the model, quantization, context size, runtime version, and hardware beside them. A tokens-per-second number without that context is not reproducible.

Check your understanding: running models locally

Check runtimes, installation, model storage, API requests, Node clients, streaming boundaries, and performance measurements.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build a local AI feature

Create a Node.js summarizer with structured output, timeouts, cancellation, fallback behavior, and focused tests.

Define the feature boundary

Add local AI as one narrow transformation with explicit input, output, timeout, and fallback behavior.

We will build a local activity summarizer.

Its input is factual application data:

```
const activity = {
  focusMinutes: 52,
  completedTasks: 3,
  interrupted: false,
}
```

Its output is one sentence plus a confidence label:

```
{
  summary: 'You focused for 52 minutes and completed 3 tasks.',
  confidence: 'high',
}
```

The original activity record remains the source of truth. The generated sentence is a convenience.

Define failure before writing the request. If Ollama is missing, the model is unavailable, the request times out, or the response fails validation, the application will create a deterministic sentence from the original fields.

This boundary keeps the product useful without AI. It also gives us an exact contract to test.

Request structured output

Use a JSON schema to constrain the model response, then validate the returned value before the application trusts it.

Asking for "valid JSON" in a prompt is weaker than providing a schema.

Ollama accepts a JSON schema in the `format` field:

```
const summarySchema = {
  type: 'object',
  properties: {
    summary: { type: 'string' },
    confidence: { enum: ['low', 'medium', 'high'] },
  },
  required: ['summary', 'confidence'],
  additionalProperties: false,
}
```

Send that schema with `stream: false`. Put the same output requirements in the prompt so the task and structure agree.

A constrained response is still untrusted input. Parse the JSON, check both fields, enforce the sentence length, and compare factual claims with the source data where possible.

A schema can guarantee shape. It cannot guarantee truth.

Use a low temperature when consistency matters, then test whether the chosen model follows the schema reliably.

Add timeout and cancellation

Stop work that has exceeded the feature budget or is no longer needed, while distinguishing cancellation from service failure.

A local request can still hang or take too long.

Pass an abort signal to `fetch()`:

```
const signal = AbortSignal.timeout(8000)

const response = await fetch('http://localhost:11434/api/chat', {
  method: 'POST',
  headers: { 'content-type': 'application/json' },
  body: JSON.stringify(request),
  signal,
})
```

Eight seconds is an example, not a universal default. Measure the real model and choose a budget that matches the feature.

Cancellation has another use. If the user closes a view or replaces the input, the old result may no longer matter. Abort the pending request so a stale answer cannot update the interface later.

Handle these outcomes separately in logs:

- user or lifecycle cancellation
- timeout
- connection failure
- HTTP error
- invalid model output

They may share the same deterministic fallback, but they need different diagnostics.

Add a deterministic fallback

Preserve the application value when local inference is unavailable by generating a truthful result from source data.

The fallback should not pretend to be another AI model.

Build the sentence directly from fields the application already trusts:

```
function fallbackSummary(activity) {
  return {
    summary: `You focused for ${activity.focusMinutes} minutes and completed ${activity.compl
    confidence: 'high',
  }
}
```

Call the model inside a narrow wrapper. Return the validated model result on success and the deterministic result on expected local failures.

Do not hide programming errors. A misspelled variable or invalid application state should fail a test, not quietly look like an Ollama outage.

The interface can label generated text when that distinction matters. It does not need to interrupt the user every time the optional enhancement falls back.

This pattern is useful for summaries, labels, suggestions, and other features where the original data remains valuable by itself.

Use tools with a permission boundary

Treat model tool calls as proposed structured requests and keep authorization, validation, and execution in application code.

Some local models can request tools.

You describe a function and its parameters. The model may return a tool call instead of a final answer. Your application decides whether to execute it.

That last sentence is the security boundary.

The model does not gain permission because it produced valid JSON. Validate the tool name and arguments, authorize the operation for the current user, apply limits, and ask for confirmation before an irreversible action.

A safe loop looks like this:

```
model proposes call
       |
       v
validate -> authorize -> confirm if needed -> execute
       |
       v
return narrow result to model
```

Start with read-only tools. A function that returns today's activity is easier to contain than one that deletes files or sends messages.

Tool output is also untrusted data. A document or web result cannot grant itself new permissions by telling the model to ignore your rules.

Keep the provider swappable

Define a small application interface so local and cloud implementations can change without leaking provider details through the product.

Do not let the entire application depend on Ollama response objects.

Define the operation your product needs:

```
type ActivitySummary = {
  summary: string
  confidence: 'low' | 'medium' | 'high'
}

type Summarizer = {
  summarize(activity: Activity): Promise<ActivitySummary>
}
```

An Ollama implementation can call the local API. A cloud implementation can call a provider. A deterministic implementation can support tests and fallback behavior.

Normalize errors and output at the adapter boundary. The UI should not need to know whether a provider calls generated text `message.content`, `output_text`, or something else.

Swappable does not mean identical. Local and cloud models can differ in capabilities, privacy, la-

tency, and cost. The interface preserves application structure while your evaluation decides whether each implementation is acceptable.

Test the local AI feature

Test deterministic behavior directly and exercise the real model with a small evaluation instead of making every test depend on local inference.

Split verification into two layers.

Unit tests should cover the deterministic parts:

- prompt construction
- schema and output validation
- timeout and error mapping
- fallback sentence
- provider adapter contract

These tests run without Ollama. They should be fast and repeatable.

Integration checks exercise the installed model. Keep a small set of representative activity records and run them against the exact model tag used by the application.

Do not assert one exact generated sentence. Check the contract: valid shape, supported facts, maximum length, acceptable latency, and no invented activity.

Record failures beside the input and model version. A later model upgrade must pass the same cases before becoming the default.

The model is nondeterministic. Your acceptance boundary should not be.

Check your understanding: building a local feature

Check feature boundaries, structured outputs, validation, timeouts, fallback behavior, tool permissions, adapters, and tests.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate local AI responsibly

Review privacy, model supply chains, permissions, upgrades, costs, evaluations, and production boundaries.

Map the complete data flow

Follow prompts, files, tool results, logs, telemetry, and backups before claiming that a local AI feature is private.

Running the weights locally removes one network dependency. It does not prove the complete application is private.

Draw every place data can travel:

```
input -> application -> local model -> output
```

```
   |          |              |          |
 logs     telemetry       tools      backups
```

A local agent may call web search. A desktop application may send crash reports. Your own application may log prompts to a hosted service. A backup may copy generated files to cloud storage.

Classify the data before choosing the model. Decide which fields may enter prompts, which must be removed, how long outputs remain, and which tools may receive derived information.

Bind a local server to the smallest network boundary possible. Add authentication and isolation when several users or devices share it.

Privacy is a property of the whole data flow, not a label attached to the model.

Verify model files and code

Treat model repositories, conversion scripts, templates, and custom runtime code as a software supply chain.

Model files come from a supply chain.

Prefer original publishers or established maintainers. Check the repository identity, source model, revision, license, conversion notes, and file hashes when available.

Safetensors avoids the arbitrary-code behavior of Python pickle files. GGUF stores tensors and metadata for inference. Safe data formats reduce one risk, but a model repository may also ask you to run custom code or installation scripts.

Be careful with options such as `trust_remote_code`. They mean you are allowing code supplied by the repository to execute. Inspect and pin that code before using it with sensitive data.

A chat template can also change behavior. A conversion that uses the wrong tokenizer or template may produce poor results even when the weight file itself is intact.

Record the artifact hash and runtime version for important deployments. If a repository changes later, you can still identify what you tested.

Pin and upgrade models

Treat a model change like a dependency upgrade with named versions, saved evaluations, staged rollout, and a rollback path.

A model name is part of application behavior.

Do not replace it casually because a newer model appears on a leaderboard. The new version may follow prompts differently, use more memory, change tool-call syntax, or regress on your exact task.

Record:

- model repository and revision
- local tag or file hash
- quantization
- runtime and version
- prompt or template version
- generation settings
- evaluation results

Run the saved evaluation before upgrading. Compare quality, latency, memory, and failure cases on the real hardware.

Roll out gradually when the feature matters. Keep the previous artifact available until the new one proves stable.

Weights give you a useful advantage here: you can preserve the exact tested artifact instead of depending on a provider to keep a remote model unchanged.

Understand the real cost

Compare local hardware, electricity, maintenance, utilization, and model quality with the complete cost of an API.

Local inference has no per-token invoice, but it is not free.

Count the hardware over its useful life, electricity while computing, idle power when the machine must stay available, storage, maintenance time, and replacement cost.

Then compare an equivalent service. A small local model should not be justified by comparing it only with the price of a much stronger frontier API.

Local often makes sense when:

- you already own suitable hardware
- data must remain inside a controlled boundary
- usage is frequent and predictable
- a modest model passes the task evaluation
- offline operation matters

An API often makes sense when usage is low or spiky, the task needs frontier quality, you do not want to operate hardware, or scaling beyond one machine matters.

A hybrid design is normal. Route private routine work locally and difficult public-data work to a cloud model when policy allows it.

Complete the local summarizer

Finish the course by documenting and testing a local structured-output feature with a deterministic fallback and explicit operating boundaries.

Complete the activity summarizer from the previous module.

Your finished project should include:

1. A documented input and output contract.
2. An Ollama adapter using one pinned local model.
3. A JSON schema and runtime validation.
4. A timeout and cancellation path.
5. A deterministic fallback.
6. Unit tests for deterministic behavior.
7. A small real-model evaluation set.
8. A short operating note.

The operating note should answer:

- Which model, quantization, runtime, and hardware did you test?

- What data enters the model and where else can it travel?
- Which failures trigger fallback?
- What quality and latency did you observe?
- Which actions can the model request or perform?
- How will you test an upgrade and roll back?

Run the project once with Ollama available. Then stop Ollama and run it again. The application should still produce a truthful summary.

The goal is not to make the model impossible to replace. The goal is to build a useful feature whose boundaries you understand.

Check your understanding: operating local AI

Check complete data flows, model-file security, reproducible upgrades, real costs, hybrid deployment, and the final project boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/local-ai-models/.