

Local LLM Applications and Python Environment Setup Guide

Category: LLM Local Model Applications

1. Chat4All (GPT4All Desktop)

Installation (Linux): Download the **GPT4All** desktop application (often referred to as Chat4All) from the official website. For Linux, an Ubuntu-compatible installer (`.run` file) is provided github.com. After downloading, make it executable (`chmod +x gpt4all-installer-linux.run`) and run it to install the app. On Windows or macOS, use the provided installer for your platform and follow the setup wizard.

Adding a Local Model: Launch the Chat4All (GPT4All) application. Initially, no model is loaded by default. To add a local LLM model:

1. Open the **Models** panel from the left sidebar (below “Chats”). Click on “+ **Add Model**”, which opens the *Explore Models* page docs.gpt4all.io.
2. Search for a model by name or browse the list. GPT4All integrates with Hugging Face’s model hub, listing compatible models (usually in **GGUF** format for llama.cpp backend) [docs.gpt4all.io](https://docs.gpt4all.io/docs.gpt4all.io).
3. Click **Download** next to the model you want. The model file (several GB in size, depending on the model) will download to your machine docs.gpt4all.io. By default, models are stored in GPT4All’s data directory (e.g., `~/local/share/nomic.ai/GPT4All` on Linux) docs.gpt4all.io, but you can change this download path in the settings.
4. Once downloaded, the model appears in your “Models” list. Select it or set it as the **Default Model** in settings to use it for chat docs.gpt4all.io.

Using Local Documents for RAG: Chat4All supports retrieval-augmented generation via the **LocalDocs** feature. Click the “**LocalDocs**” button (top-right of the chat window) to open the document panel docs.gpt4all.io. Here you can **add local files** (PDF, TXT, Markdown, etc.) into a document collection. The application will index these documents by splitting them into snippets and creating embeddings for each snippet. When you ask a question, GPT4All will retrieve relevant snippets from your files and include them in the context it gives the model docs.gpt4all.io. This means you can query your own documents privately. For example, you might drop in a PDF and then ask the chat about its content – the answer will cite or draw from the relevant parts of your file. *By default, GPT4All indexes text from files with extensions `.txt`, `.pdf`, `.md`, etc., and will retrieve up to a few snippets per query* [docs.gpt4all.io](https://docs.gpt4all.io/docs.gpt4all.io). (These settings can be adjusted in the **LocalDocs Settings**).

Key Settings and Parameters: In the **Settings** menu, you’ll find important configuration options for Chat4All:

- **Context Length:** This is the maximum token length of the prompt + response that the model can handle. GPT4All (Chat4All) default is 2048 tokens for the input contextdocs.gpt4all.io. If you use LocalDocs, the retrieved snippets count toward this limit. A larger context allows longer conversations or bigger documents, but using a model with too long input may slow down generation.
- **Max Response Length:** You can set the maximum tokens for the model's answer. By default it may be 1024 or 2048 tokens, but you can adjust this as needed.
- **Temperature:** This controls the randomness of the model's output. Lower values (e.g. 0.2) make replies more deterministic and focused, while higher values (e.g. 0.8) produce more varied or creative answersdocs.gpt4all.io. The default temperature is around 0.7docs.gpt4all.io. For factual Q&A, a lower temperature might be preferable, whereas for creative writing you might increase it.
- **Model File/Location:** In *Application Settings*, you can see or change the **Download Path** for modelsdocs.gpt4all.io. By default, on Linux this is `~/ . local/share/nomic.ai/GPT4All`, on Windows it's under your AppData folder, and on macOS under `~/Library/Application Support/nomic.ai/GPT4All`docs.gpt4all.io. The downloaded model files (with extension `.gguf` or similar) are stored here. You can move models to a custom location if needed (and update the path in settings).
- **Device / CPU Threads:** Chat4All can run on CPU by default (and on GPU if using a supported backend like Nomic Vulkan). In Settings you can choose the **Device** ("Auto", "CPU", "GPU", or Apple "Metal" on M1/M2 Macs)docs.gpt4all.io. You can also set the number of **CPU threads** to use for inferencedocs.gpt4all.io – increasing this can speed up responses if you have multiple cores, but setting it too high may make your system lag.

After configuring a model and settings, you can start a new chat. Type your question or prompt and the local model will respond. The chat interface supports features like chat history, and even follow-up suggestions if enabled (the app can suggest follow-up questions based on the conversation)docs.gpt4all.io. All computation is done locally – no API keys or internet required.

2. LM Studio

Installation (Linux): Download **LM Studio** from the official website. On Linux it's distributed as an AppImage (a portable Linux binary). Choose the Linux version of LM Studio from the download page (supports x64 and ARM64)[apidog.com](https://lmstudio.ai). Save the `.AppImage` file (e.g., `LMStudio-x64.AppImage`) to your system. To run it, first make the file executable:


```
chmod +x LMStudio-x64.AppImage
```

Then launch it:

```
./LMStudio-x64.AppImage
```

This will start LM Studio; no installation to the system directories is needed (the AppImage runs in place). *Tip:* You can optionally move the AppImage to `/usr/local/bin` or another location in your PATH for easier access [apidog.com](https://lmstudio.ai). LM Studio is available for Windows and macOS as well (as a typical `.exe` installer or `.dmg` package). On first launch, no login is required – LM Studio can be used entirely offline.

Downloading and Configuring a Model: LM Studio provides a convenient GUI to find and install local models:

1. Use the **Discover** tab (the model search interface) to find models. You can click the  icon or press `Ctrl+2` to open the model search in LM Studio lmstudio.ai. Type keywords or model names in the search bar (for example, “Llama 2 13B”). LM Studio will display matching models from Hugging Face Hub that are compatible (GGUF or similar formats).
2. Click on a model result to see download options. Models often have multiple quantization variants (like `Q4_K_M`, `Q8_0`, etc.) – these refer to different precision levels of the model weights lmstudio.ai. A **lower-bit quantization (e.g. 4-bit)** uses less memory at some cost to quality, whereas 16-bit (or higher bit) versions are larger but more accurate. Choose a variant that fits your hardware (the UI may show RAM requirements).
3. Click **Download** on your chosen model file to begin downloading. The model will be fetched from Hugging Face and cached locally. You can monitor progress in the interface. Once done, the model will appear in the **My Models** section. By default, LM Studio stores models in its application data directory, but you can change the models directory in settings if desired lmstudio.ai.
4. **Load the model:** After downloading, go to the **Chat** tab or **Local Server** tab in LM Studio. In the chat interface, select the model from the dropdown at the top before starting a conversation. If using the Local Server feature (to serve the model via API), select the model and click “Load” or “Start Server”. If the model fails to load (e.g., out of memory), you might need to choose a smaller model or quantization that your system can handle.

Local Document Ingestion (RAG): LM Studio supports retrieval-augmented generation via its *Chat with Documents* feature. In an active chat, you can attach documents to supplement the AI’s context. Click the **paperclip (attach)** icon in the chat window (or drag-and-drop a file into the chat). LM Studio accepts PDF, text, or Word documents (`.pdf`, `.txt`, `.docx`) lmstudio.ai. When you attach a document, LM Studio will analyze its content in one of two ways:

- If the document is **short enough to fit in the model’s context** entirely, LM Studio will directly insert the full text into the conversation (preceding or alongside your query) lmstudio.ai. This is useful for small reference texts or prompts.
- If the document is **too long**, LM Studio will automatically switch to **Retrieval-Augmented Generation (RAG)** mode lmstudio.ai. It will index the document’s content (likely by splitting into chunks and embedding them behind the scenes) and then, when you ask a question, it retrieves the most relevant snippets. Those snippets are provided to the model as context,

instead of the entire document. This way, even large documents can inform the model's answers without exceeding the context window.

When using this feature, you might, for example, load a 50-page PDF and ask the model a question about it. The model will “read” it by retrieving relevant parts rather than the whole PDF at once. **Tip:** To improve retrieval accuracy, make your question as specific as possible – mention key terms or sections you think are relevant. This helps the system pick the right snippetslmstudio.ai. (RAG isn't always perfect, so a bit of user guidance can go a long way in getting a good answer.)

Key Notes: LM Studio's interface also includes settings for things like model **GPU acceleration** and an **OpenAI-compatible local API**. On capable hardware, LM Studio can use your GPU to accelerate inference (this is often enabled by default if it detects an appropriate GPU). It also has a **Local Server** tab where you can start an OpenAI-style API on a port (default 1234) to serve the model – useful if you want to connect external applications similar to how one would with Ollama or an OpenAI API. In summary, LM Studio provides an all-in-one solution: a chat UI, model downloader, and optional API server, all running locally and offline.

3. Ollama

Installation:

- **Linux:** Open a terminal and run the official one-line installer for Ollama:

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

This script downloads the latest Ollama binaries and sets up the service. After it completes, you should have the `ollama` command available. (Under the hood, it also registers a systemd service called `ollama.service` to run the Ollama server in the background on startup[centron.de](https://centron.de/centron.de).) You can verify the installation by running `ollama -v` to get the version[centron.de](https://centron.de/centron.de). The Ollama service typically starts automatically upon install; check `sudo systemctl status ollama` to ensure it's active (it should show Ollama running/active)[centron.de](https://centron.de/centron.de).

- **macOS:** Download the **Ollama for macOS** package from the official website. It comes as a ZIP or DMG. For example, you might get a `Ollama.app.zip`. Unzip it and drag **Ollama.app** into your **Applications** folder[centron.de](https://centron.de/centron.de). The first time you launch Ollama.app, it will ask for confirmation (since it's downloaded from the internet). Once opened, Ollama runs a background service (and adds the `ollama` CLI to your PATH). You can then use the `ollama` command in Terminal. Check by opening Terminal and running `ollama -v` or `ollama list`[centron.de](https://centron.de/centron.de) to confirm it's set up.
- **Windows:** Download the Windows installer (`OllamaSetup.exe`) from the Ollama site. Run the installer and follow prompts (no admin rights required, it installs to your user profile). After installation, Ollama will be running in the background (it adds a tray icon) and the `ollama` CLI will be available in Command Prompt or PowerShell[docs.ollama.com](https://docs.ollama.com/docs.ollama.com). You can open PowerShell and run `ollama -v` to confirm. On Windows, Ollama listens on the

same default localhost port (11434) and stores model data in your user directories (check %HOMEPATH%\ .ollama for models)docs.ollama.com. *Note:* For Windows with GPUs, ensure you have the latest NVIDIA or AMD drivers if you plan to use GPU accelerationdocs.ollama.com.

Installing and Running Models Locally: Once Ollama is installed, using it involves CLI commands:

- **Pulling a model:** Ollama has a built-in model registry. You can download (pull) models by name. For example, to get qwen2.5-coder:1.5b you might run:

```
ollama pull qwen2.5-coder:1.5b
```

In general, use `ollama pull <model_name>` to download a modelcentron.de. There are many models available (Ollama has its own repository of supported models; these include variants like `mistral`, `stablelm`, `phi`, etc., and community models). For instance, to download the **Mistral 7B** model, you would run `ollama pull mistral`centron.de. To get a specific version or quantization, some models have tags (for example `ollama pull deepseek-r1:1.5b` would pull a 1.5B-parameter distilled model)centron.de. After pulling, use `ollama list` to see all models you have downloaded locallycentron.de – it will list model names, IDs, and sizes.

- **Running a model (CLI):** To run a model and chat via the terminal, use `ollama run`. For example:

```
ollama run qwen2.5-coder:1.5b
```

This will load the Llama2 model and drop you into an interactive prompt session. You can then type a question or prompt, hit Enter, and the model will output a response in the terminal. It works like a REPL for the AI. For instance, you could run `ollama run mistral` and then at the `>>>` prompt, type *“Hello, how are you?”* and the model will reply. The session continues until you exit. (To exit an interactive session, type `>>> /bye` on a new line and press Entercentron.de.) You can also have Ollama execute a one-off prompt by using the `--prompt` flag or echoing text into it, but interactive mode is straightforward for chatting.

Another example:

```
ollama run qwen2.5-coder:1.5b
```

This would load the Qwen 2.5 model (1.5B parameters version) and let you interact with itcentron.de. Make sure your system has enough memory for the model you choose; if not, the `ollama run` command may fail to load the model (in which case try a smaller model or a more quantized version).

- **Model files:** Ollama stores models in a directory (on Linux/macOS, it's `~/ .ollama` by default; on Windows, `%HOMEPATH%\ .ollama`). Model files can be large (several GB). Keep an eye on disk space. If needed, you can move the models directory by setting an environment variable `OLLAMA_MODELS` to a new pathdocs.ollama.com (do this **before** pulling models).

Connecting Ollama to a Chat Interface: One of Ollama’s strengths is its built-in API server. When the Ollama service is running (it usually runs automatically in the background as a system service or via the app), it exposes an **OpenAI-compatible HTTP API** at `http://localhost:11434` docs.ollama.com. This means you can use third-party chat UIs or tools that normally talk to the OpenAI API, and point them to your Ollama server.

For example, **Open WebUI** (an open-source chat interface) supports integration with local Ollama models. In Open WebUI, you can add a custom OpenAI endpoint and set the URL to `http://localhost:11434` (with no API key needed) docs.openwebui.com. This will let you chat with the models you have in Ollama through a nice web browser interface, complete with message history, etc. Similarly, developer tools like the VS Code extension “Continue” or chatbots like SillyTavern can connect to Ollama by treating it as if it were the OpenAI API – you just configure the endpoint URL to the local server and specify the model name.

If you prefer not to keep the server running at all times, you can manually start it: run `ollama serve` in a terminal to start the API server process foreground docs.ollama.com apidog.com. Once running, it listens on port 11434 by default. You can then send API calls to it.

Serving via API (using Ollama): The API is similar to OpenAI’s REST API. For instance, you can do a POST request to `http://localhost:11434/api/generate` with a JSON body like:

```
{
  "model": "qwen2.5-coder:15.b",
  "prompt": "Why is the sky blue?",
  "stream": false
}
```

and you’ll get a response with the model’s answer. Here’s an example using PowerShell to call the API locally docs.ollama.com:

```
(Invoke-WebRequest -Method POST -Body '{"model":"llama2","prompt":"Why is the sky blue?","stream":false}' -Uri http://localhost:11434/api/generate).Content |
ConvertFrom-Json
```

This would return a JSON response containing the generated text. In a similar way, you could use `curl` or Python’s `requests` library to hit the endpoint. By default, no authentication is needed for local access (since it’s on your machine). Ollama’s API supports both chat-style and completion-style interactions (the exact format may differ slightly from OpenAI’s, but it’s documented in Ollama’s docs). Using the API, you could build a small web app, integrate the model into a script, or connect a GUI. For a more interactive use, it’s often easiest to use an existing UI (like the aforementioned Open WebUI or the LM Studio’s client with “Use MCP” for Ollama).

In summary, **Ollama** provides a robust way to manage and run local models via CLI and exposes an API for integration. You install it once, then simply `ollama pull` models you need, and you can chat either in terminal or through any tool that can speak to its local API. All computation stays on your machine.

Category: Non-LLM Local Environment & Interfaces

4. Python Environment Setup

When working on local AI projects, it's good practice to isolate your Python dependencies in a virtual environment. Let's create a new virtual environment called **workshop_env** for our project.

Step 1: Create the virtual environment. In a terminal, run:

```
python3 -m venv workshop_env
```

This uses Python's built-in venv module to create a folder `workshop_env` containing a clean Python interpreter and site-packages directory github.com. (You might need to use `python` instead of `python3` on Windows, depending on how Python is installed. Ensure you have Python 3.9+ for best compatibility github.com.)

Step 2: Activate the environment. Activation means your shell will start using the `workshop_env`'s Python and pip.

- On **Linux/Mac**, activate by running:

```
source workshop_env/bin/activate
```

This changes your `$PATH` to use the env's Python. You should see your prompt prefixed by `(workshop_env)` indicating it's active.

- On **Windows (cmd.exe)**, run: `workshop_env\Scripts\activate.bat`. For **PowerShell**, use: `workshop_env\Scripts\Activate.ps1`. Once activated, the prompt on Windows will also show `(workshop_env)`.

After activation, running `python --version` or `which python` should point to the Python inside `workshop_env` (and pip will install into this environment).

Step 3: Install required packages. With the env activated, install the packages you need using pip. For example, the user needs `fastai`, `gradio`, `transformers`, and `fastbook`. Install them all at once:

```
pip install fastai gradio transformers fastbook
```

This will download the latest versions from PyPI. These libraries are: **FastAI** (a high-level deep learning library), **Gradio** (a web UI library for demos), **Transformers** (Hugging Face's NLP library), and **fastbook** (notebooks and utilities from the *FastAI* book/course). The packages might have additional dependencies which pip will handle automatically.

During installation, you'll see pip resolving dependencies and then downloading wheels. Once complete, you can verify by launching a Python REPL (`python`) and trying `import fastai, gradio, transformers, fastbook`. If no errors, the install is successful. All these packages are now installed **only in workshop_env**, not globally, which keeps your system Python clean.

Using the Environment: Whenever you work on this project, activate the `workshop_env` first. If you open a new shell, don't forget to run the `source workshop_env/bin/activate` (or the Windows equivalent) again, otherwise you'll be using system Python without those libraries. If you're done or want to exit the environment, just run `deactivate` and your shell will return to normal (this is a shell command provided by `venv`). The environment folder `workshop_env` can be copied or moved with the project (though on Windows you shouldn't move it across drives). In case you need to recreate it (say you messed up packages), you can always delete the folder and make a new `venv`, then re-install the packages with `pip`. It's also a good idea to freeze the requirements (`pip freeze > requirements.txt`) for reproducibility, but that's optional for this guide.

5. Windsurf Setup

Installing Windsurf (Linux): Windsurf is an AI-powered IDE (from the makers of Codeium) that is built on a VS Code base. To install **Windsurf on Linux**, download the Linux **Windsurf Editor** AppImage from the official site apiidog.com. The AppImage is a self-contained package. Suppose you downloaded `Windsurf-x64.AppImage` to your `Downloads` folder. First, make it executable:

```
chmod +x ~/Downloads/Windsurf-x64.AppImage
```

Then run the AppImage:

```
./Windsurf-x64.AppImage
```

This will launch the Windsurf IDE. *(No root installation is needed, but you can optionally integrate it with your desktop environment by creating shortcuts or moving the AppImage to a convenient location.)* On first launch, Windsurf might prompt you to log in to your Codeium/Windsurf account to enable cloud features or analytics – you can create a free account if you want, or skip if offline. Once Windsurf opens, it will also initialize its language servers. Being based on VS Code, it may detect existing VS Code settings or extensions on your system and import them apiidog.com (for example, if you had the Python extension in VS Code, it could carry over). You can also manually install extensions in Windsurf as needed, via its Extensions panel (it has its own marketplace).

Connecting to `workshop_env`: Since Windsurf is essentially VS Code under the hood, you connect it to a Python virtual environment the same way you would in VS Code. Open Windsurf and **open your project folder** (or any folder) where you want to work. Make sure the **Python extension** is enabled (Windsurf should have it, as it's a common extension; if not, install the official Microsoft Python extension through the Extensions marketplace). Now, select the Python interpreter for your project: In the bottom-left corner of the window you might see an indicator that says `Python 3.x` or **Select Interpreter**. Click that, and you'll get a list of available Python environments. You should see an entry for **`workshop_env`** (it will show the path, e.g. `.../workshop_env/bin/python`). Choose that interpreter. Alternatively, press **Ctrl+Shift+P** to open the Command Palette and start typing "Python: Select Interpreter", then select the `workshop_env`'s Python code-literacy.medium.com. Once selected, Windsurf will use that environment for running code, IntelliSense, etc.

You can verify it's using the right one by opening a new Terminal within Windsurf (it will typically activate the environment automatically in a new integrated terminal session) or by running a quick `which python` (it should point to the path inside `workshop_env`). Now your IDE is set to use the `workshop_env` virtual environment, meaning any code you run or debug will use the packages we installed earlier (fastai, transformers, etc.).

Calling a Hugging Face Model in Windsurf: With everything set up, let's do a quick test to ensure we can call a local model from within Windsurf. We'll use the Hugging Face **Transformers** library pipeline. Open a new Python file in Windsurf (e.g., `test_pipeline.py`) and enter the following code:

```
from transformers import pipeline

# Create a text-generation pipeline with a small model
generator = pipeline("text-generation", model="distilgpt2")
result = generator("Hello, I am a local model,")[0]["generated_text"]
print(result)
```

When you run this script (you can use the play button or F5 to run with debugging, or simply right-click and "Run Python File"), the Transformers library will automatically download the **DistilGPT2** model the first time github.com. This is a tiny GPT-2 model, so it should download quickly. The code then uses the model to generate a continuation for the prompt "Hello, I am a local model," and prints it. You should see an output string in the terminal/console of Windsurf. Each run after the first will use the cached model from disk (no re-download).

How it works: The Hugging Face pipeline API makes it super easy to use models. In our code, `pipeline("text-generation", model="distilgpt2")` downloads the `distilgpt2` model (if not already present) and sets up a text generation pipeline github.com. We then call `generator(prompt)` to generate text. The result comes back as a list of dictionaries; we take the first item `[0]` and grab the `"generated_text"`. The output might be something like: *"Hello, I am a local model, and I will continue this sentence with something somewhat coherent."* (Each run may produce a different continuation because of randomness in text generation). This confirms our environment is working: the `transformers` library is installed and could fetch a model, and we executed it all within Windsurf.

Because Windsurf is an IDE with AI features, you can also leverage its AI coding assistant (Cascade) to help write code that uses these libraries. For instance, you could prompt Windsurf's AI chat to "Write a short script that uses Hugging Face Transformers pipeline for sentiment analysis," and it should auto-generate code (since it now also knows about your environment's imports). The combination of a local environment (`workshop_env`) and Windsurf's AI suggestions can accelerate development while keeping everything offline.

Finally, remember that to use the `workshop_env` in future sessions, you'll need to ensure Windsurf has that interpreter selected each time (it usually remembers per workspace). If you create new virtual environments for other projects, you'd go through the same selection process. This ability to connect IDE to specific Python envs allows you to manage dependencies per project smoothly.

