ChatGPT

LM Studio Python SDK – Comprehensive Overview

Overview of LM Studio and its SDK

LM Studio is a user-friendly desktop application for running and experimenting with large language models (LLMs) locally on your computer [1]. It supports downloading and running a wide range of open-source models (e.g. LLaMA, Mistral, Qwen, etc.) in **GGML/GGUF** formats, with GPU acceleration when available [2]. LM Studio provides a familiar chat interface and tools for managing models, and it can operate as a local LLM server exposing OpenAI-like APIs [3] [4].

To enable programmatic access to local models, the LM Studio team provides a **Python SDK** (officially called `lmstudio-python`). This SDK allows developers to interact with the LM Studio backend from Python code, enabling tasks like loading models, generating text completions or chat responses, retrieving embeddings, and even defining tools for autonomous agent-like behaviors – all running **completely on local hardware** [5]. In essence, the SDK lets you integrate LM Studio's LLM capabilities into your own Python applications without using cloud APIs.

Documentation and Repository

The official documentation for LM Studio and its SDK is available on the LM Studio docs site. The Python SDK has a dedicated section with guides and examples [6]. The source code is open-source (MIT licensed) and hosted on GitHub under the **lmstudio-ai** organization: the repository **"lmstudio-python"** contains the SDK's code and examples [7]. This repository is the best source for code reference and issue tracking (there is also a separate bug-tracker repo for the LM Studio app itself [8]).

Community resources: For help and discussion, LM Studio maintains an active Discord community [9] where users and developers share knowledge. Additionally, developers often discuss LM Studio on forums like Reddit (e.g. the r/LocalLLaMA subreddit) and have created integration projects (for example, a LlamaIndex connector and LangChain how-to guides) to use LM Studio in broader AI workflows. The LM Studio team also publishes updates on their blog (e.g. release notes for new versions) and interacts via GitHub issues for feature requests or known issues.

Installation and Setup of the Python SDK

The LM Studio Python SDK is distributed via PyPI. You can install it with a simple pip command:

```
pip install lmstudio
```

This will install the package `lmstudio` in your environment [10]. The SDK supports Python 3.10 and above (classified for 3.10, 3.11, 3.12, 3.13) [11]. It's recommended to use a modern Python (the SDK is regularly updated to add features and may drop very old Python versions in major releases [12]).

Prerequisites: Since the SDK is a client interface to the LM Studio LLM service, you should have the **LM Studio application** installed on your machine. LM Studio is available for macOS (Apple Silicon), Windows (x64/ARM64), and Linux (x64) [13] [14]. Ensure that you have run LM Studio at least once to complete initial setup (this creates the necessary folders and installs the CLI tool). On first launch, you can download an LLM model through the GUI or the CLI.

Running the LM Studio server: The SDK communicates with a local server component of LM Studio. By default, LM Studio runs a background server on the loopback interface (usually port 1234). If you launch the LM Studio app's **Developer** or **Server** mode (or enable "run LLM server on login" in settings [15]), the server will listen for API requests. You can also start it manually via the CLI:

```
lms server start
```

[16]. Once the server is running, the Python SDK can connect to it. (By default it assumes `localhost:1234` unless configured otherwise.)

Custom host/port: If you are running the LM Studio server on a non-default port or a different machine, you can configure the SDK accordingly. Before making any SDK calls, set the default client's address:

```python
import lmstudio as lms
lms.configure_default_client("your-hostname:port")
```

This must be done *before* any other SDK call (to avoid the SDK auto-initializing with the default) [17] [18]. For example, if the server runs on a remote machine or a custom port, supply `"192.168.1.10:1234"` or similar. (In SDK versions <1.3.0, the method was `lms.get_default_client(host_port)` to achieve the same [19].)

Authentication: *By default, LM Studio's local API does not enforce authentication.* All requests to the local server are accepted without an API key, which is convenient for local use. However, if exposing the service over a network, you might need to implement your own security (the developers have discussed adding API key support for security [20], but typically the expectation is you run it on `localhost` or behind a firewall). When reusing OpenAI client libraries (discussed below), you can supply any token (e.g. `"lm-studio"`) as a dummy key if the client requires one [21] – LM Studio will ignore it by default.

Basic Usage: Initializing the SDK Client

When you import and use `lmstudio`, the SDK under the hood manages a **WebSocket connection** to the LM Studio server. The core object is a `Client` that handles this connection [22]. For convenience, the SDK provides a *global client* so you often don't need to explicitly create one – simply using top-level API functions will initialize a default client in the background [23].

There are two approaches to using the SDK:

- **Convenience API (global client):** High-level functions that implicitly use a default `Client` instance. This is great for interactive use (e.g. Jupyter notebooks or quick scripts) since you can call functions directly. The default client stays active until the Python process exits [24].

- **Scoped API (explicit client):** You can create a `Client` manually and manage its context (e.g. using `with` statements to ensure proper closure). This is useful in larger applications where you need fine control over resource lifecycle (ensuring sockets close deterministically) [25]. The SDK documentation provides a parallel example for most operations using the explicit client (`with lms.Client() as client: ...`) if needed.

For getting started, the convenience API is simplest. For example, to obtain a handle to a language model, use the `lms.llm()` function:

```
import lmstudio as lms
model = lms.llm()
```

If called with no arguments, `lms.llm()` returns a handle to the currently loaded model (if one is already active in memory) [26]. If no model is loaded yet, calling `lms.llm("model-key")` with a specific model identifier will attempt to load that model into memory (or return the existing instance if already loaded) [27]. The `"model-key"` is typically the model's name as listed in LM Studio (for example, `"llama-3.2-1b-instruct"` or an identifier you've assigned).

Example – Loading a specific model:

```
model = lms.llm("qwen2.5-7b-instruct")
```

This will either retrieve the Qwen 2.5 (7B instruct) model if it's already loaded, or initiate loading it from disk into memory [28]. (Under the hood, this is equivalent to using the client's `load_model` or similar API. It provides *Just-In-Time* loading behavior, so you don't have to manually preload models in most cases.)

Downloading models: Note that the SDK does *not* handle downloading model weights – it assumes models are present on your system. You can download models through the LM Studio GUI or using the CLI: for example, `lms get <model-name>` will fetch a model from Hugging Face [29]. Ensure you run that beforehand if you don't already have the model files. (The CLI `lms ls` command will list downloaded models, and `lms ps` lists currently loaded models [30] [31].)

Model management: The SDK provides functions to list and unload models as well. For instance, `lms.list_downloaded_models()` and `lms.list_loaded_models()` can enumerate models on disk and in memory respectively. Each model handle (the object returned by `lms.llm()`) has an `unload()` method to free it from RAM when no longer needed [32]. Advanced usage even allows multiple instances of the same model loaded under different identifiers (using `load_new_instance`) if you want to run them in parallel [33] [34]. Typically, though, you will load one model at a time due to resource constraints.

Generating Text: Chat and Completion Workflows

Once you have a `model` handle, you can generate text in two modes:

- **Chat Completions (multi-turn conversations)** – The model treats input as a dialogue with roles (system, user, assistant). Use the `respond()` method.

- **Text Completions (single-turn prompts)** – The model completes a given text prompt. Use the `complete()` method.

Chat Completions (Conversational AI)

For chat use-cases, LM Studio's SDK provides a convenient `Chat` context object. A `Chat` represents the conversation history (system instructions + user messages + assistant responses). You can create a chat and add messages to it, then ask the model to generate the next reply.

Basic example (single prompt):

```python
import lmstudio as lms
model = lms.llm()  # use default loaded model
response = model.respond("What is the meaning of life?")
print(response)
```

In this simple form, passing a string to `respond()` implicitly treats it as a user message (with no prior context) [35] [36]. The model will produce a single-turn answer.

Multi-turn conversation example:

```python
model = lms.llm()
chat = lms.Chat(system_message="You are a helpful shopkeeper.")
chat.add_user_message("My hovercraft is full of eels!")
response1 = model.respond(chat)  # assistant responds to user
chat.add_assistant_response(response1)
print("Assistant:", response1)

chat.add_user_message("I will not buy this record; it is scratched.")
response2 = model.respond(chat)
print("Assistant:", response2)
```

In this example, we initialized a `Chat` with a system role instruction and then added two user messages sequentially [37] [38]. After each user query, we call `model.respond(chat)` to get the assistant's reply, and optionally record it back into the chat history (`add_assistant_response`) for the next turn. The `Chat` object automatically formats the conversation into the model's expected input format. (Under the hood, LM Studio applies a prompt template appropriate to the model family, so the roles are understood correctly [39].)

Streaming responses: The SDK can stream token-by-token output, which is useful for responsiveness or for printing results as they come. Instead of `respond()`, use `respond_stream()`:

```python
stream = model.respond_stream(chat_or_prompt)
for fragment in stream:
    print(fragment.content, end="", flush=True)
print()  # newline after completion
```

Each `fragment` yielded is a piece of text (or message) from the model [40] [41]. This allows your application to display or process partial results (e.g., updating a UI in real-time). Once the stream is exhausted, you can obtain the full result and metadata via `result = stream.result()` [42].

Controlling generation: You can pass a `config` dictionary to `respond()` or `respond_stream()` to customize inference parameters (very similar to OpenAI's parameters). For example:

```
prediction = model.respond(chat, config={
    "temperature": 0.6,
    "maxTokens": 50,
    "stopStrings": ["\nUser:"]
})
```

This would generate with a slightly lower temperature and limit the response to 50 tokens [43]. Supported parameters include temperature, max tokens, presence/frequency penalties, stop sequences (`stopStrings`), top-k/top-p sampling settings, etc., as documented in the **"Configuration Parameters"** section [44]. If you omit config, default model settings or UI-set presets will apply.

Retrieving metadata: After generation, the result object contains useful info. For instance, `result.model_info` gives the model name, and `result.stats` includes metrics like token counts, time to first token, and the stop reason [42]. You can log or display these to evaluate performance (e.g., tokens per second).

Cancellation: If needed, the SDK supports cancelling an ongoing prediction (for example, if the user stops the request). This is covered under "Cancelling Predictions" in the docs [45] and is typically done by calling a cancel method on the client or stream handle.

Advanced chat features: LM Studio introduced support for *function calling* or "tools" where the model can output a JSON calling a function and the SDK can execute it. In the SDK, this is handled via the `.act()` method and tool definitions (enabling agentic flows) [46] [47]. You can define Python functions as tools, and the model can invoke them in the conversation (completely offline, similar to OpenAI function calling but local). This is an advanced topic – see the "Agentic Flows" section of the docs for details.

Text Completions (Prompt Completion)

If you want the model to simply complete a piece of text (like autocompleting a sentence or code generation), you can use `model.complete(prompt)`:

```
model = lms.llm()
result = model.complete("Once upon a time,")
print(result)
```

This will produce a continuation of the given text [48]. It's analogous to OpenAI's `/v1/completions` endpoint. You can also stream completions with `complete_stream()` similarly to chat. The config options and usage pattern are the same (pass a `config` dict if needed, and iterate over the stream for tokens).

The SDK differentiates chat vs completion primarily by the method you call. Internally, LM Studio will apply any prompt formatting for chat (like inserting system/user role tokens) when using `respond()`, whereas `complete()` sends the text raw to the model [49] [50].

For example, if you have a large prompt (like a user's entire console history or story beginning) and want the model to continue it, `complete()` is appropriate. After getting the result (or stream), you can again access `result.content` for the text and use `result.stats` for performance info [51] [52].

Example: The documentation provides a creative use-case where `complete()` is used to simulate a terminal session. It continually feeds the model a history of user commands and model outputs, using a stop token (`"$"` prompt) to stop generation at the next command prompt [53] [54]. This demonstrates how you can craft completion prompts for non-chat scenarios as well.

Embeddings and Tokenization

In addition to generating text, LM Studio's SDK can generate text embeddings and tokenize text using local models:

- **Embeddings:** If you have a vector embedding model (for example, text embedding models), you can use `lms.embedding_model("model-key")` to load it, then call `.embed(text)` to get an embedding vector. The LM Studio docs show how to generate embedding vectors for input text [55] [5] (embedding support may depend on models that are designated as embedding models). The OpenAI compatibility API also supports the `/v1/embeddings` endpoint [56] [57], so you could use either approach to generate embeddings locally for semantic search or similar tasks.

- **Tokenization:** You can access the tokenizer of a loaded model via the SDK. For instance, the docs indicate you can tokenize text and even detokenize if needed [58] [55]. Functions like `model.tokenize("text")` might be available, or a separate utility in the SDK. This is useful to check token counts or for splitting inputs to fit model context limits. (Each model has a context length, which you can query via `model.context_length` or a dedicated call [59].)

Keep in mind that tokenization is model-specific (different models have different vocabularies and context lengths), so always use the tokenizer of the same model you plan to use for inference.

Integration with Python Applications

One of LM Studio's strengths is that it can act as a drop-in local replacement for OpenAI's API. There are a few ways to integrate LM Studio-hosted models into applications:

1. Using the LM Studio Python SDK directly: This is ideal for Python applications where you control the code. As shown above, you can load models and get completions in just a few lines of Python. This approach gives you full access to LM Studio's features (like advanced config, streaming, etc.) natively.

2. OpenAI API Compatibility: LM Studio exposes REST endpoints that mimic OpenAI's API structure [60]. The default base URL is `http://localhost:1234/v1`. You can take any existing code that uses OpenAI's official SDK and point it to LM Studio. For example, using OpenAI's Python client:

```python
import openai
openai.api_base = "http://localhost:1234/v1"
openai.api_key = "lm-studio"  # dummy key
response = openai.ChatCompletion.create(
    model="your-model-id",
    messages=[{"role": "user", "content": "Hello"}]
)
```

By changing the `api_base` (or `base_url`) to your LM Studio server and using the model identifier that you have loaded in LM Studio, the rest of the OpenAI call works as usual [61] [21]. This means you can integrate with frameworks like **LangChain** or others that expect an OpenAI-like interface simply by configuring the base URL. (LangChain's `ChatOpenAI` class, for instance, can accept a `openai_api_base` parameter to redirect to LM Studio, and then you can use local models in LangChain flows.) Many community guides demonstrate this – e.g., using LM Studio with LangChain to build a private Q&A bot by just swapping the API endpoint.

3. LM Studio REST API (enhanced): In addition to pure OpenAI compatibility, LM Studio provides a REST API with additional endpoints and stats (in beta) [62] [63]. This includes endpoints to list all *downloaded* models (not just loaded ones) and provides detailed timing information (tokens/sec, etc.). A Python application could directly call these via HTTP (using `requests` or similar). However, since the Python SDK internally already uses the WebSocket API to communicate, you typically wouldn't need to call the REST API yourself – the SDK is a more convenient wrapper. The REST API is useful if you are integrating from a non-Python language or want to build a small service without adding the Python SDK dependency.

4. CLI-based automation: For simple scripting or non-Python contexts, remember that the `lms` CLI can be invoked to manage models and even perform prompt inference. For instance, `lms load <model>` and `lms unload` can be called from shell scripts, and `lms server start/stop` to control the server. There are also community tools (like browser extensions or low-code platforms) that interface with LM Studio by calling its OpenAI-style API or using the CLI in the background.

Integration libraries: The community has created specialized connectors. One example is the **llama-index-llms-lmstudio** package [64] which integrates LM Studio with LlamaIndex (GPT Index) – you start the LM Studio server, then use their `LMStudio` class in place of an OpenAI LLM within LlamaIndex [65] [66]. This shows how easy it is to incorporate LM Studio: essentially point to `base_url="http://localhost:1234/v1"` and specify the model name. Similarly, for LangChain, you can use `OpenAI` or `ChatOpenAI` classes with the base URL override, or use LangChain's **LocalAI** integration if available. The key is that **any tool expecting an OpenAI API can be configured to use LM Studio**, allowing you to leverage local models in existing AI applications (chatbots, agents, etc.) without heavy code changes.

Compatibility and System Requirements

Supported platforms: LM Studio (and thus its SDK) works on modern macOS (Apple Silicon), Windows, and Linux systems [13] [67]. Notably, macOS requires Apple Silicon (Intel Macs are currently not supported [68]) and Linux support is x86_64 only (no ARM yet) [69]. Windows support includes x64 and upcoming ARM64 (Snapdragon) PCs [14]. The SDK itself is pure Python and should run wherever Python runs, but it needs to connect to an LM Studio server – so effectively, you need LM Studio installed on a compatible machine.

Python versions: As mentioned, the SDK targets Python 3.10+ [11]. It likely works on 3.9 in practice (there are reports of requiring >=3.9 for some integrations [70]), but to use the latest version, upgrade to 3.10 or newer. The project uses semantic versioning for the SDK, and they bump the major version if they drop older Python support or make breaking changes [71]. Always check the PyPI release notes or documentation for any version-specific considerations.

Hardware requirements: Running large models locally can be resource-intensive. While not a limitation of the SDK itself, you should be mindful of system specs: - **RAM:** 16GB or more is recommended [72] [73], especially for models 7B and above. Smaller models (2-3B) can run on 8GB but with limited context. The SDK will happily try to load a model even if it might swap – it's up to the user to ensure the system can handle it. - **VRAM:** If you have a GPU, LM Studio can offload model layers to it (the CLI and SDK allow specifying `gpu` usage percentage on model load [74]). A GPU with 4GB+ VRAM is recommended to see speed benefits [73]. The SDK's default `llm()` load will use whatever GPU offload is configured by your LM Studio runtime settings. - **CPU:** LM Studio uses efficient backends like llama.cpp and Apple's core ML (MLX) for Apple Silicon. On x86, AVX2 support is required for CPU inference [67]. Most modern CPUs have this, but older ones might not – in which case models won't run. Ensure your CPU is AVX2-capable.

Software dependencies: The `lmstudio` PyPI package will install required Python dependencies automatically (like WebSocket clients, etc.). The heavy lifting (model math) is done by LM Studio's runtime (C++ backend) which is part of the LM Studio app installation. So you don't need PyTorch or TensorFlow in your Python environment for this SDK – it delegates computation to the LM Studio server.

Limitations and Known Issues

While LM Studio and its SDK are powerful, you should be aware of some current limitations and issues:

- **Feature Maturity:** LM Studio is under active development (versions have rapidly evolved through 0.2.x to 0.3.x in late 2023 and early 2024). Some features are beta or evolving. For example, *auto-unloading* of models (to free VRAM/RAM after inactivity) was not available until recently – as of v0.3.5, models loaded via API remain in memory until explicitly unloaded, though a TTL auto-evict feature was introduced in 0.3.9 [75]. Always refer to the latest docs for such updates.

- **Closed Source Runtime:** The LM Studio application is (currently) closed source. This means the core execution backend isn't directly inspectable or modifiable by the community. However, the SDK and CLI are MIT-licensed, and many models it supports are open source. The closed nature mainly impacts how quickly community can debug deeper issues. That said, the developers are responsive on Discord and GitHub for bug reports.

- **API Differences:** The OpenAI-compatible API aims to mirror OpenAI's responses, but there could be minor differences. For instance, model IDs in LM Studio are not the same as OpenAI's (you use your local model's name). Also, not all OpenAI endpoint features may be present. (Function calling in OpenAI's sense isn't directly an API field yet, although LM Studio's own tooling system covers similar ground in a different way.) If your application relies on very specific OpenAI API behaviors, test carefully with LM Studio's API.

- **No Internet / Tools (by default):** Local models are sandboxed – if your workflow expects the AI to browse the web or fetch information, note that LM Studio doesn't provide internet access to models out-of-the-box. You could implement tools in the SDK (e.g. a tool that performs web search when the model calls it via `.act()`), but that requires custom setup. Similarly, “plugins” in the ChatGPT sense aren’t a thing here, aside from user-defined tools.

- **System Limitations:** As mentioned, support for certain systems is lacking (no native Intel Mac support yet, etc.). On Windows, some users have reported stability issues or the need to update GPU drivers for best performance. Always check the **System Requirements** doc and the latest release notes for any platform-specific caveats.

- **Large Model Limitations:** Very large models (e.g. 70B with high context length) can be slow or even not load if you don’t have enough RAM. The SDK will return errors if a model fails to load due to memory. It’s often useful to use quantized models (like 4-bit quantizations) to reduce memory, which LM Studio supports (the model catalog in the app often provides GGUF quantized variants).

- **Known bugs:** For the latest known issues, refer to the official bug tracker on GitHub [8]. For example, as of early 2025 there were reports of UI crashes on certain Windows setups [76] and some mathematical reasoning inaccuracies in specific models [77] (these are usually model-specific). The developers encourage bug reports to improve the product.

In summary, while LM Studio offers a cutting-edge way to run LLMs locally, it’s still a fast-evolving project. Always keep your SDK and app updated to benefit from improvements (e.g., performance boosts like the integration of NVIDIA’s CUDA acceleration in v0.3.15 [78]).

Community and Support

Official support: The primary support channel is the LM Studio **Discord server**, where you can ask questions and get help from both the development team and a growing user community [9]. They have channels for troubleshooting, sharing prompts, and discussing new features. The developers are quite active there.

Forums and Q&A: Users often discuss LM Studio on Reddit (for example, in subreddits focused on local LLMs). Common topics include how to fix connectivity issues, comparisons between LM Studio and other local LLM runners (like Ollama or text-generation-webui), and tips for optimizing model performance. If you encounter an issue, it’s worth searching Reddit or Stack Overflow – e.g., questions like *“Error connecting to LM Studio”* have been asked by others [79].

GitHub: The LM Studio team has multiple repositories on GitHub (for the SDK, CLI, etc.) and an issue tracker for the app. If you find a reproducible bug or have a feature request, you can open an issue. For instance, suggestions like adding API key authentication have been discussed on the GitHub issue tracker [80]. There’s also a **changelog** in the docs for the API and app, detailing new features and fixes in each release [81].

Learning resources: A number of tutorials and walkthroughs are available to help new users: - *Official docs & guides:* The documentation includes “how-to” sections, such as **Chat with Documents** (demonstrating retrieval-augmented generation entirely offline) and **Import Models** (for adding custom or private models). These are great for learning advanced use-cases through step-by-step instructions. - *Video tutorials:* There are YouTube videos like **“How to Run LLM Models Locally with LM Studio”** that provide a visual step-by-step guide (covering installation, configuration, and usage). Another popular one is **“Run ANY Open-Source LLM Locally (No-Code LMStudio Tutorial)”**, which is often recommended for beginners [82] [83]. These videos show the process of setting up LM Studio, downloading a model, and making your first queries, all without writing code. - *Articles and blogs:* Several AI enthusiasts have written blog posts on using LM Studio. For example, a Medium article by Yuwei Sung walks through enabling developer mode and using the OpenAI-compatible server to send prompts via a local API [84] [85]. Others have written guides on integrating LM Studio into applications (e.g., building a private Q&A bot with LM Studio + LangChain + Chroma for retrieval [86]). These community tutorials can provide context, troubleshooting tips, and ideas for projects. - *Courses:* There's even a beginner-friendly course on Udemy ("LM Studio for Beginners: Run LLMs locally") [87], indicating the demand for structured learning on this tool.

In short, you are not alone in exploring LM Studio – an entire community is experimenting with running LLMs locally. Make use of these resources to deepen your understanding and quickly resolve any issues you encounter.

Conclusion

LM Studio's Python SDK empowers developers to harness powerful large language models on local hardware with relative ease. To recap, you install the SDK with pip, ensure the LM Studio server is running, and then you can load models and generate text or embeddings with simple Python calls. It integrates smoothly with existing workflows (thanks to OpenAI API compatibility) and supports a variety of advanced features like streaming, tool use, and customization of inference parameters. While there are some limitations inherent to running models locally, the rapid development of LM Studio (and its growing model catalog) makes it a compelling solution for private, offline LLM applications.

By following the official documentation [10] [48] and community guides, you can quickly get started with LM Studio's SDK – whether you aim to build a local chatbot, experiment with prompts, or integrate an LLM into your software. As always, keep your setup updated and engage with the community to get the most out of what LM Studio has to offer. Happy coding with your local LLMs!

Sources:

- LM Studio Documentation (Python SDK, API, and App Guides) [10] [88] [61] [28]
- LM Studio GitHub Repositories (lmstudio-python SDK and CLI) [89] [90]
- Community Tutorials and Resources (Medium article, integration examples, NVIDIA blog) [84] [65] [78]
- LM Studio System Requirements and Release Notes [91] [75]
- LlamaIndex LMStudio integration (example usage) [65] [66] and OpenAI API compatibility guide [21].

---

[1] [3] [4] [9] [13] About LM Studio | LM Studio Docs
https://lmstudio.ai/docs/app

[2] [82] [83] LM Studio: A User-Friendly LLM Application - Datatunnel
https://datatunnel.io/tldr_listing/lm-studio-a-user-friendly-llm-application/

[5] [6] [7] [10] [25] [29] [46] [47] [55] [58] [89] lmstudio-python (Python SDK) | LM Studio Docs
https://lmstudio.ai/docs/python

[8] [15] [16] [75] Run LM Studio as a service (headless) | LM Studio Docs
https://lmstudio.ai/docs/app/api/headless

11 22 23 lmstudio · PyPI
https://pypi.org/project/lmstudio/

12 24 37 38 48 71 GitHub - lmstudio-ai/lmstudio-python: LM Studio Python SDK
https://github.com/lmstudio-ai/lmstudio-python

14 67 68 69 72 73 91 System Requirements | LM Studio Docs
https://lmstudio.ai/docs/app/system-requirements

17 18 19 Project Setup | LM Studio Docs
https://lmstudio.ai/docs/python/getting-started/project-setup

20 21 39 56 57 60 61 OpenAI Compatibility API | LM Studio Docs
https://lmstudio.ai/docs/app/api/endpoints/openai

26 27 28 32 33 34 59 Manage Models in Memory | LM Studio Docs
https://lmstudio.ai/docs/python/manage-models/loading

30 31 74 90 lms — LM Studio's CLI | LM Studio Docs
https://lmstudio.ai/docs/cli

35 36 40 41 42 43 44 45 88 Chat Completions | LM Studio Docs
https://lmstudio.ai/docs/python/llm-prediction/chat-completion

49 50 51 52 53 54 Text Completions | LM Studio Docs
https://lmstudio.ai/docs/python/llm-prediction/completion

62 63 LM Studio as a Local LLM API Server | LM Studio Docs
https://lmstudio.ai/docs/app/api

64 65 66 70 llama-index-llms-lmstudio · PyPI
https://pypi.org/project/llama-index-llms-lmstudio/

76 LM Studio crashes : r/LocalLLaMA - Reddit
https://www.reddit.com/r/LocalLLaMA/comments/1ltqf9a/lm_studio_crashes/

77 Issues · lmstudio-ai/lmstudio-bug-tracker - GitHub
https://github.com/lmstudio-ai/lmstudio-bug-tracker/issues

78 LM Studio Accelerates LLM With GeForce RTX GPUs | NVIDIA Blog
https://blogs.nvidia.com/blog/rtx-ai-garage-lmstudio-llamacpp-blackwell/

79 lm studio not working - General - oTTomator Community
https://thinktank.ottomator.ai/t/lm-studio-not-working/527

80 API keys to secure the endpoint · Issue #561 · lmstudio-ai ... - GitHub
https://github.com/lmstudio-ai/lmstudio-js/issues/202

81 API Changelog | LM Studio Docs
https://lmstudio.ai/docs/app/api-changelog

84 85 Serving LLM locally using LM Studio | by Yuwei Sung | Medium
https://yuweisung.medium.com/serving-llm-locally-using-lm-studio-c83baa552768

86 Quick and Dirty: Building a Private RAG Conversational Agent with ...
https://medium.com/@mr.ghulamrasool/quick-and-dirty-building-a-private-conversational-agent-for-healthcare-a-journey-with-lm-studio-f782a56987bd

87 LM Studio for Beginners: Run LLMs locally - Udemy
https://www.udemy.com/course/lm-studio-for-beginners/?srsltid=AfmBOoof9eYy53UtRVCkPEFx8gyd6VbDtwx1AcGITc175CEKK4kyxZ3o