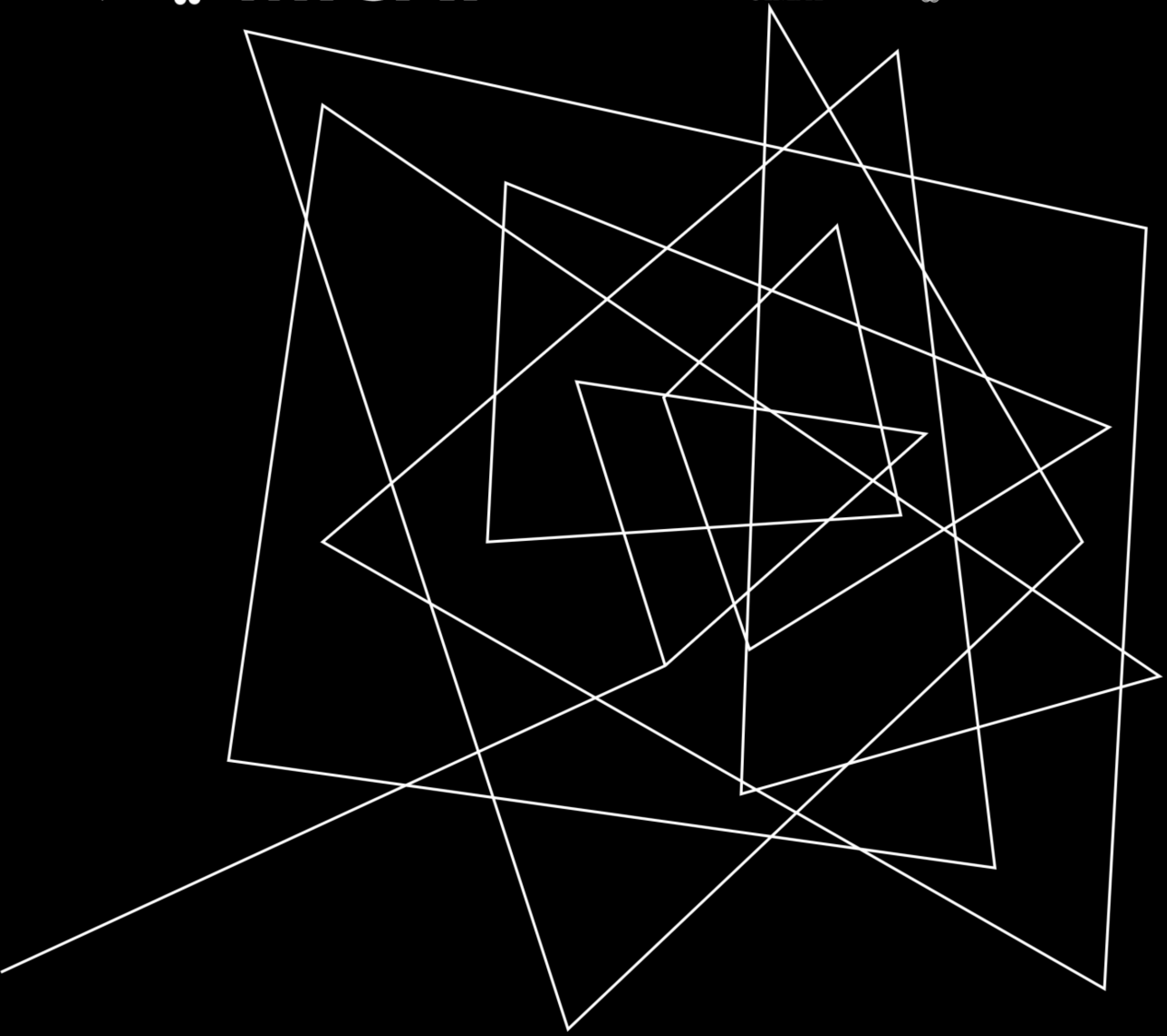


BEYOND COMMERCIAL APIS: HOSTING OPEN LLMS FOR YOUR NEEDS

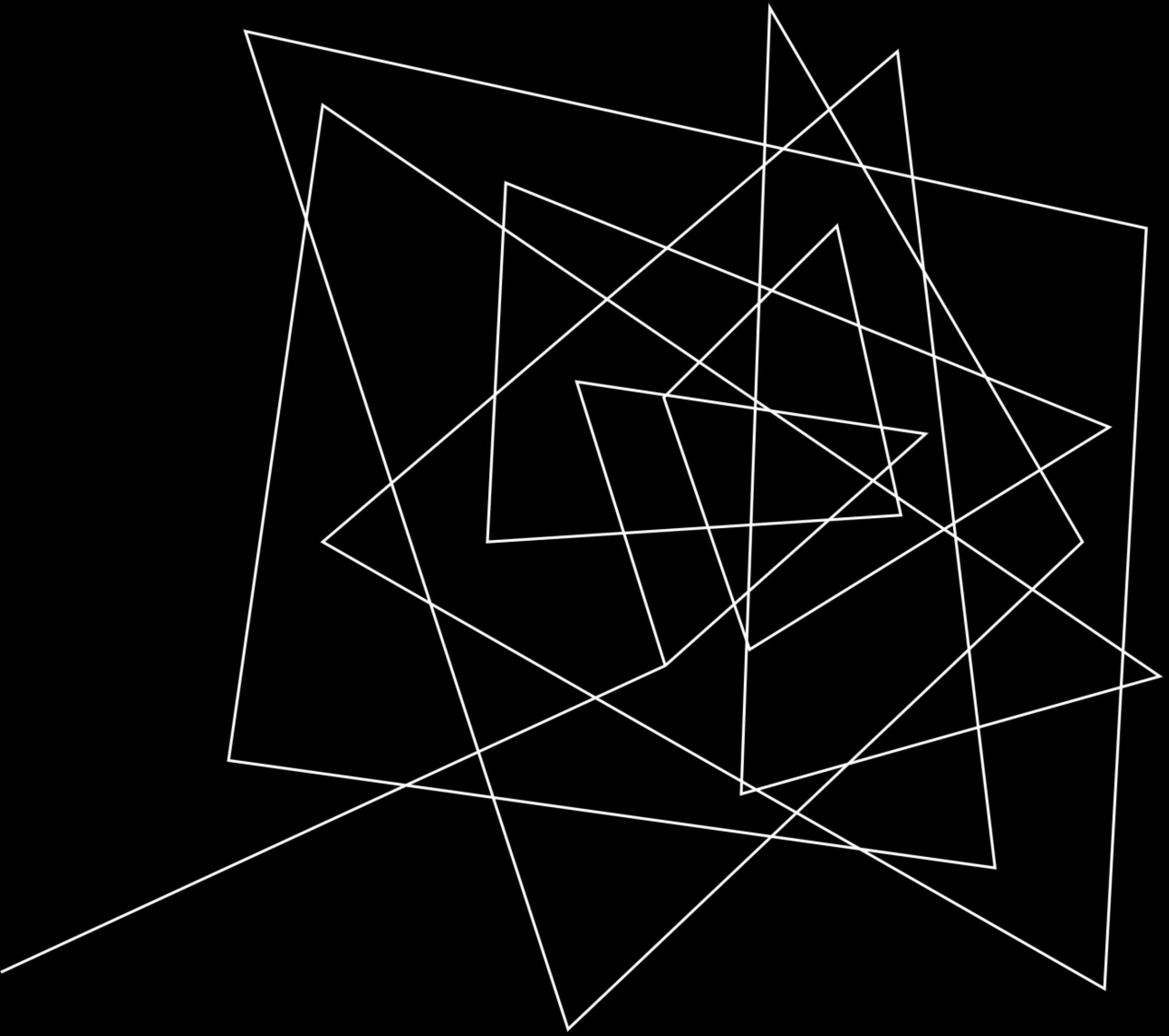
Cesar Torres*, Claudia I. González

* cesartorres@tectijuana.mx

November-2025

Agenda

- Introduction
- Open LLM Landscape
- Serving Backends
- Hardware and Optimization
- Front-Ends
- Integration and Apis
- Full Hands-On Workflow
- Best Practices and Trends
- Q&A



Before we start

Before We Start

You can Download the slide deck from here:

https://github.com/CESARIUX2596/beyond-commercial-apis-llm-tutorial

Before We Start

Accounts

- 🐙 GitHub(https://github.com/): for code access and issue tracking.
- 🤗 Huggingface (https://huggingface.co/): Download models and datasets, create an account

Technologies:

- Miniconda: https://www.anaconda.com/docs/getting-started/miniconda/install#quickstart-install-instructions
- Ollama: https://ollama.com/
- Postman: https://www.postman.com/downloads/
- WSL (Ubuntu 24.04) +

Repos to check/use:

- VLLM: https://github.com/vllm-project/vllm
- OpenWebUI: https://github.com/open-webui/open-webui
- Stable-diffusion-webui-forge: https://github.com/lllyasviel/stable-diffusion-webui-forge

Before We Start

What we will use (start downloading before starting the hands-on-tutorial):

- Qwen3-0.6B

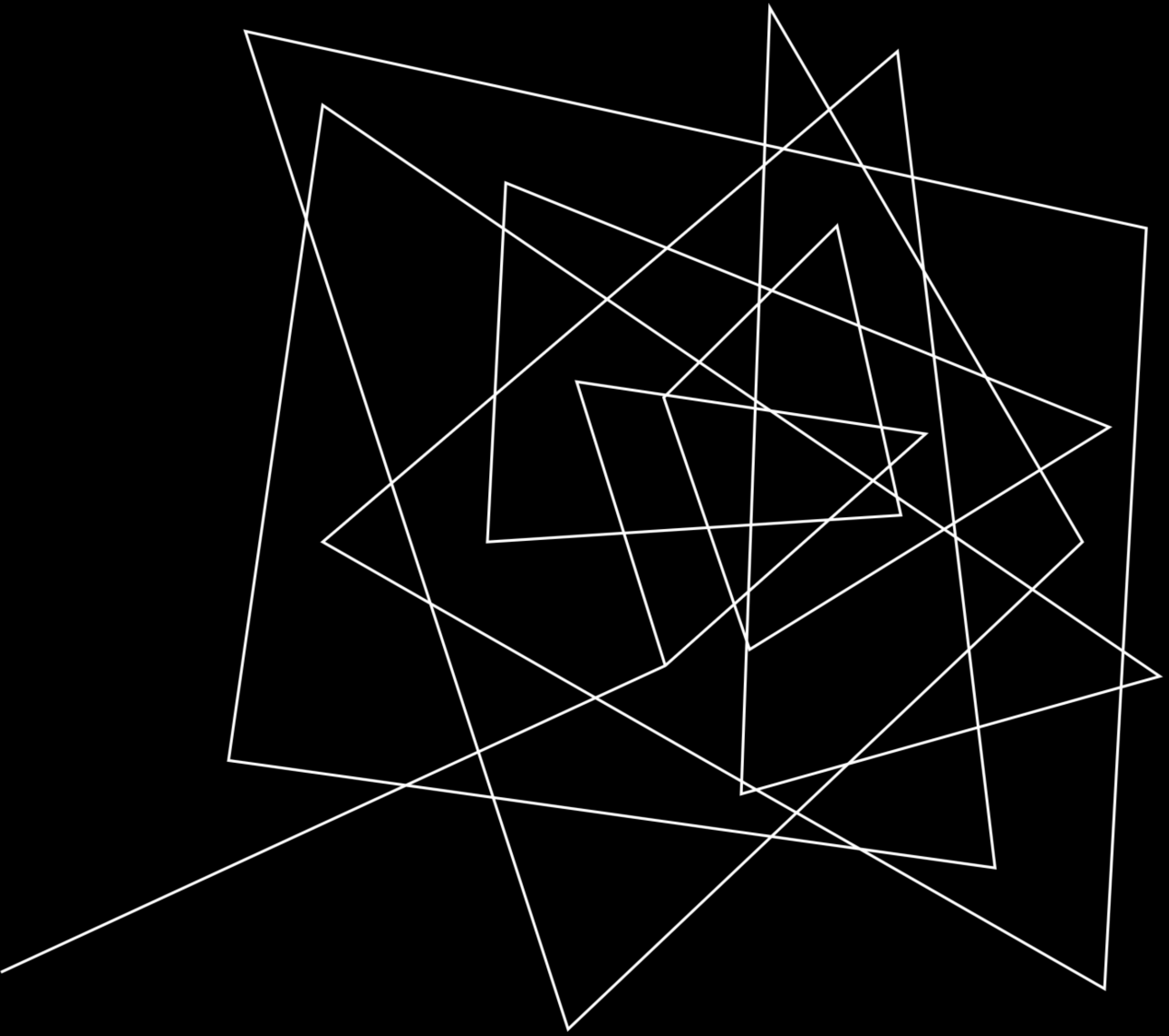
  https://huggingface.co/Qwen/Qwen3-0.6B

- Llama-3.1-8b-Instruct

  https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

- Stable-diffusion-v1-5 (6-12GB of Vram) [Nvidia only at the moment]

  https://huggingface.co/stable-diffusion-v1-5/stable-diffusion-v1-5



Introduction

Who I Am and What I Work On

- Software Developer @ Thermo Fisher Scientific (Internal AI Platform).
- PhD Candidate in Machine Learning at TECNM / Tijuana Institute of Technology.
- Former Machine Learning Engineer @ Samsung Research Tijuana (SRT).
- Research areas: Generative AI, Computer Vision, Multimodal Learning, Self-Hosted Systems.
- Advocate for Linux and open infrastructure.

My Research at a Glance

Generative AI & LLMs

- Domain Adaptation.
- Efficient Fine-tunning (LoRA, QLoRA).
- Prompting.

Computer Vision & Multimodality

- Visual Transformers.
- Image Synthesis.

Self-Hosted AI Systems

- vLLM, Ollama, MS-Swift.
- Edge inference and private cloud hosting.

A simple diagram showing your *research workflow pipeline*:
Data → Model → Fine-Tuning → Local Deployment → Applicationcc

Why Go Beyond Commercial APIs?

Commercial APIs

- Expensive at scale.
- Data leaves your organization.
- Limited customization.
- Vendor lock-in.

Self-Hosted AI Systems

- Free to run once hosted.
- Full local privacy.
- Full control over prompts & models.
- Portable, reproductible setups.

What You’ll Learn in This Tutorial

- 🧠 Understand open LLM families & how to choose the right one.
- ⚙️ Deploy models with backends like vLLM, Ollama, and MS-Swift.
- 💻 Integrate your hosted models into applications.
- 🧩 Build simple custom front-ends.
- 🔋 Optimize performance & resource usage.

Visual:

Timeline or roadmap bar: *Intro* → *Models* → *Serving* → *Front-Ends* → *APIs* → *Demo* → *Best Practices.*

LLM Lingo

- **Parameters**: The “neurons” or tunable weights that store knowledge. More params = more memory, more nuanced reasoning.
- **Context Window**: How many tokens (words/subwords) the model can “see” at once. Longer context = handle longer inputs or conversations.
- **Tokens**: The units of text the model processes (≈ words or pieces). Used for counting cost and context usage.
- **KV Cache**: Key-Value cache — stores previous attention states during chat. Speeds up multi-turn interactions but uses VRAM.
- **Quantization**: Compressing model weights (e.g., 16-bit → 4-bit, exponential growth). Reduces memory use; slight quality trade-off.
- **Checkpoint / Weights**: The saved state of a trained model. You load this into vLLM, Ollama, etc.
- **Fine-tuning / LoRA**: Additional training to specialize a model. Adapts models for domains (biomedical, legal, etc.).
- **Serving**: Running a model as an API or local service. What we’ll be doing later!
- **Inference**: Generating text from a prompt (as opposed to training). Core activity when “hosting” a model.

LLM Types, Formats & Architectures

- **Model Types**
  - **Dense:** All parameters active per token (e.g., LLaMA, Qwen, Phi).
  - **MoE (Mixture of Experts):** Only a few experts activate per token → higher capacity, lower compute (e.g., Mixtral, DeepSeekMoE).
- **Parameter Counts**
  - Model names often include size (e.g., Llama-3-8B, Qwen2-72B) indicating billions of parameters that define capacity and hardware needs.
- **File Formats**
  - `.safetensors:` secure, fast weight loading (modern standard).
  - `.bin / .pth:` raw PyTorch checkpoints.
  - `.gguf / .ggml:` quantized formats for lightweight or local inference (used in llama.cpp, ollama, etc.).
- Weights & Variants
  - Full precision: FP32 / BF16 for training.
  - Quantized: INT8 / INT4 for faster inference.
  - LoRA / Delta (“thin”) weights: lightweight adapters for fine-tuning or specialization.

What Exactly Is a Large Language Model?

A Large Language Model (LLM) is a neural network (based in the transformer architecture) trained on vast amounts of text to predict the next word (token) in a sequence.

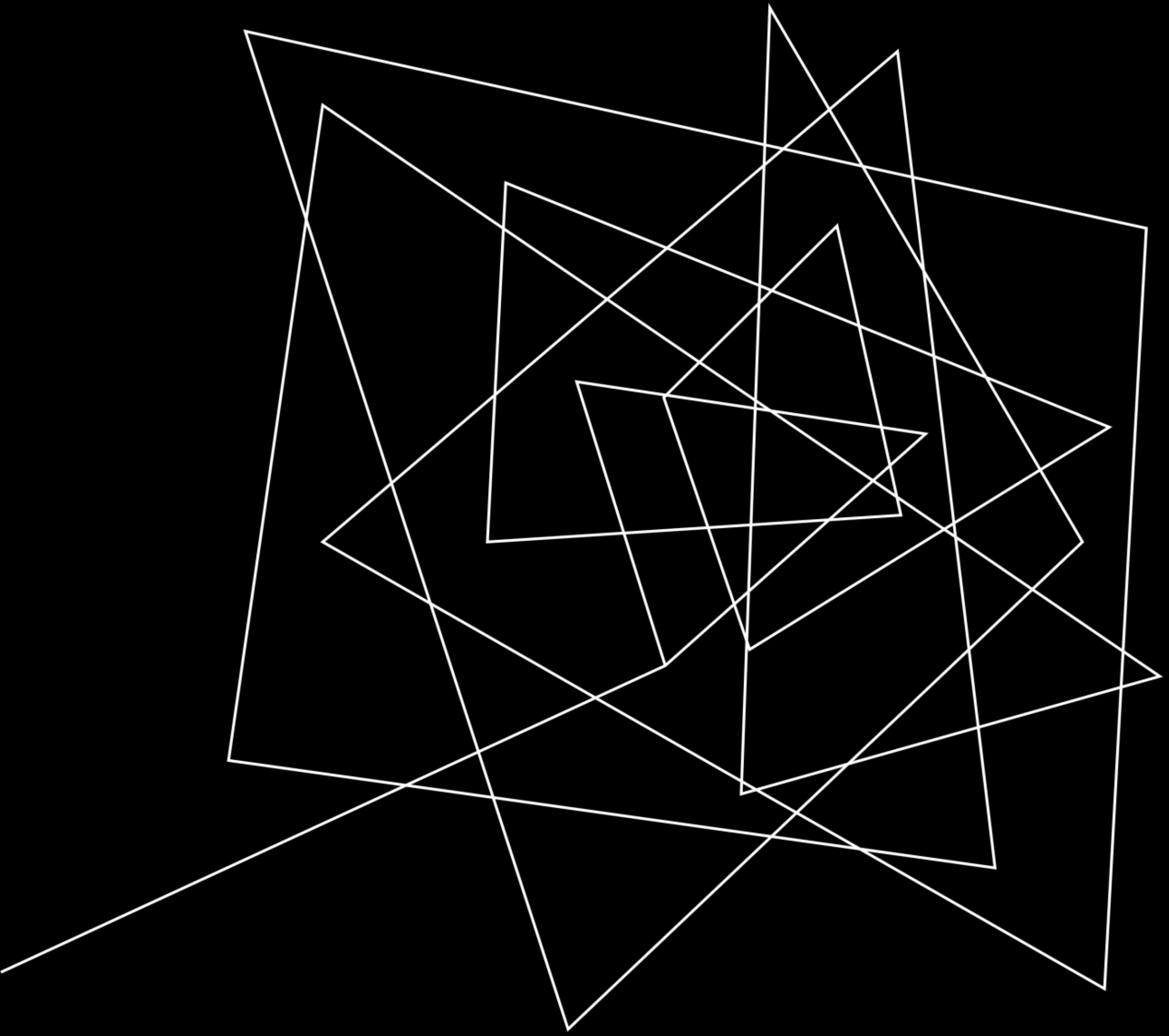
Over time, it learns grammar, facts, reasoning patterns, and context, allowing it to generate coherent, human-like text.

Main components:

- 🧮 Parameters: Billions of “weights” that encode knowledge
- 📚 Training Data: Books, websites, papers, code.
- 🗣️ Capabilities: Text generation, reasoning, summarization, dialogue.

Diagram:

Input Text → [Transformer Layers] → Predicted Next Token → Output Text



Open LLM Landscape

What Is an Open LLM?

- Open LLM = Open Weights + Open Inference Stack + Usable License.
- Usually distributed through Hugging Face or ModelScope.
- Community-driven → transparent training data and evaluation.
- Enables local serving, fine-tuning, and custom deployment.

Families of Open Models

Familly	Parameters	Context Len	License	Highlights
LLaMA 3-4	1B – 400B	Up to 256K	Meta Community / Commercial-Friendly	Broad ecosystem, Strong reasoning, Extensive fine-tune support Mixture-of-Experts (MoE).
Mistral	7B/8x7B(Mixtral)	32K	Apache-2.0	Fast inference, MoE
Qwen 3	0.6B – 235B	32K – 128K	Apache 2.0	Multilingual, Vision, Agentic support, Advanced MoE design.
Gemma	2B – 7B	8k – 32K	Apache-2.0	Google-backend small model great efficiency/low-resource performance.
Phi 3	3.8B	8K	MIT	Lightweight & efficient, Compact with strong reasoning
DeepSeek R1	7B-32B	64K	Custom Permissive	High-quality reasoning, Reinforcement-enhanced alignment.
GPT-OSS	20B	16K-32K	Apache-2.9	Open GPT-3-class model, Strong generalization, Reproducible training.

Understanding Model Types

- **Base ← general pretrained model**

  Trained on massive text corpora to learn general language patterns.

  Used as a foundation for further fine-tuning or adaptation.

- **Instruct ← task following**

  Fine-tuned on instruction–response datasets to follow human prompts.

  Optimized for single-turn tasks and factual completions.

  (Example: Llama-3-Instruct, Qwen-Instruct)

- **Chat ← conversational alignment**

  Built on instruct models with multi-turn dialogue data and safety alignment (RLHF/DPO).

  Handles conversation flow, memory, tone, and user intent better.

  (Example: Llama-3-Chat, Mistral-Chat)

The Evolution of Open LLMs (2023 → 2025)

2023 A New Hope

- **LLaMA 1 → 2** : Sparked the open-model revolution.
- **Falcon 40B**: First permissive large model.
- **Mistral 7B/Mixtral 8x7B**: efficient open challenger.

2024 The Open Strikes Back

- **Gemma (2B/7B)**: efficient Google models.
- **Qwen 2 / 3**: multilingual + multimodal.
- **Phi 3**: small yet highly capable
- **LLaMA 3**: GPT-3.5-class quality.
- **DeepSeek-R1**: open reasoning model.

2025 Return of the Models

- **LLaMA 4**: open text-vision hybrid MoE.
- **Mistral Next / Mixtral 8×22B**: large MoE evolution.
- **Gemma 2 / Qwen 3**: mature multimodal systems.
- **GPT-OSS**: community-trained GPT-4-level release.

Model Sizes in Practice: Tiny → Small → Medium → Large

- Tiny (<1B Params)
  - Use: on-CPU demos, edge devices, simple assistants.
  - Pros: runs almost anywhere, super low latency.
  - Limits: shallow reasoning, short inputs/outputs.
- Small (2B-4B Params)
  - Use: local chat, basic summarization/classification.
  - Pros: fits in low-VRAM GPUs/quatized CPUs.
  - Limits: may struggle with complex multi-step tasks.

Model Sizes in Practice: Tiny → Small → Medium → Large

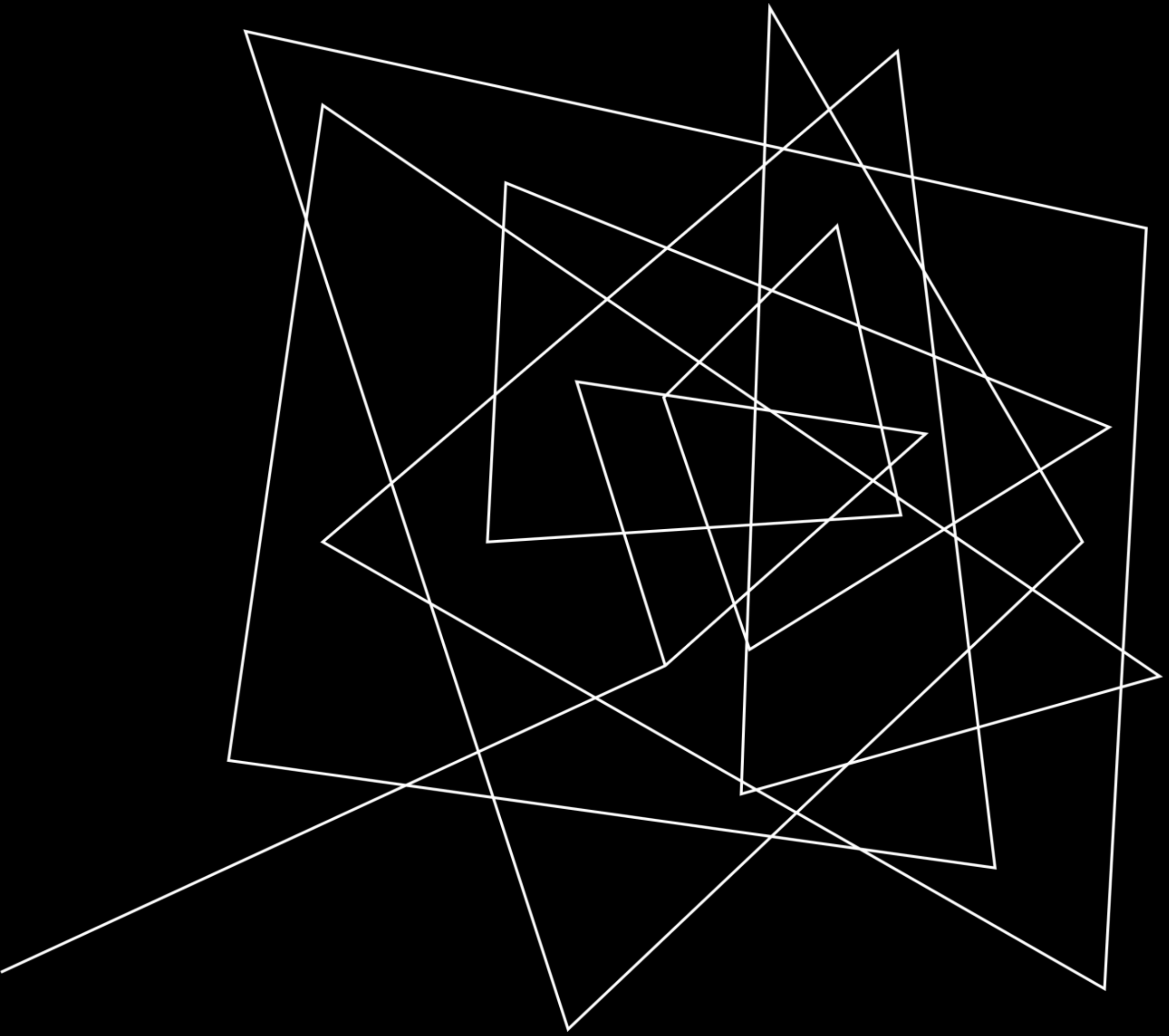
- Medium (6B-8B Params)
  - Use: general chat, RAG, decent reasoning.
  - Pros: strong quality/speed balance for laptops/desktop GPUs, easy to train.
  - Limits: long contexts and code-gen abilities, still not “pro” level.
- Upper-Mid(10B-14B Params)
  - Use: better summarization quality and decent reasoning skills.
  - Pros: noticeable jump in coherence and instruction following.
  - Limits: needs more VRAM; quantization often required.

Model Sizes in Practice: Tiny → Small → Medium → Large

- Large (20B-40B Params)
  - Use: Deep reasoning, document analysis, research QA.
  - Pros: strong coherence, handles long multi-step reasoning.
  - Limits: slow inference, requires high-VRAM GPUs, slow quantized CPU perfonmance.
- Massive (100B-400B+ Params)
  - Use: advanced reasoning, multi-agent chains, creativity.
  - Pros: GPT-4-class outputs, long effective context window (100k+).
  - Limits: needs distributed clusters, high cost/power draw.

Model Sizes in Practice: Tiny → Small → Medium → Large

- 🧩 More Parameters → better reasoning, memory, nuance.
- 🧠 More Memory → needed for larger model weights and context (KV cache).
- 🕒 Trade-off: Quality ↑ ⇒ Speed ↓ ⇒ Cost ↑.
- ⚙️ Quantization: reduces memory load (e.g., 16 GB → 6 GB) with ~1–3% accuracy loss.



Serving Backends

Engines Behind the Scenes: How We Serve LLMs

Serving backend = software that loads an LLM, manages memory, handles batching, and exposes it via an API or local interface.

Backend	Highlights	Best For	Trade-Offs
Ollama	Simplest local host, Mac/Windows/Linux ready	Desktop demos, small teams	Less flexible for scaling
vLLM	Fast OpenAI-compatible API, optimized GPU paging	Production-grade deployments	Needs CUDA GPU; setup via CLI
MS-Swift	Microservice-oriented, supports distributed setups	Multi-node servers / research labs	Newer project, evolving rapidly
Llama.cpp	C/C++ lightweight runtime, runs anywhere (CPU, Metal, CUDA)	Cross-platform, low-VRAM systems, embedded use	CLI-focused; fewer API features
LM Studio / Open-WebUI	GUI-based frontends with built-in serving	Non-technical users, testing	Limited backend control

Engines Behind the Scenes: How We Serve LLMs

- 🧩 Choose by deployment style:
  - vLLM → high-performance API.
  - Ollama / llama.cpp → personal or lightweight use.
  - MS-Swift → multi-node or distributed lab setups.
- ⚡ vLLM = fastest on GPUs; llama.cpp = most portable.
- 🧠 GUI options (LM Studio, Open-WebUI) make great teaching tools.

Under the Hood: What Model Serving Actually Does

When we serve a model, we turn a static checkpoint into a responsive system that can answer many users at once.

A modern backend usually handles five things:

1. **Loading & Initialization**. Keeps model weights, tokenizer, and quantized layers resident in memory so every prompt doesn’t reload gigabytes of data.
2. **Request Queueing & Batching**. Manages many simultaneous prompts, combining them into batches to keep the GPU fully utilized and latency low.
3. **Caching & Streaming**. Maintains each conversation’s KV cache (short-term memory) and streams tokens back in real time so responses appear instantly.
4. **API Interface & Decoding Control**. Exposes an OpenAI-compatible API (/v1/chat/completions) and handles generation parameters like temperature, top-p, and repetition penalty.
5. **Resource & Monitoring Layer**. Tracks GPU/CPU usage, releases memory pages, logs token throughput and latency, keeping deployments stable and observable.

Under the Hood: What Model Serving Actually Does

Your App / Client

↓

API Layer (OpenAI Format)

↓

Serving Core (Batching + Cache + Sampling)

↓

Hardware (GPU / CPU)

Under the Hood: What Model Serving Actually Does

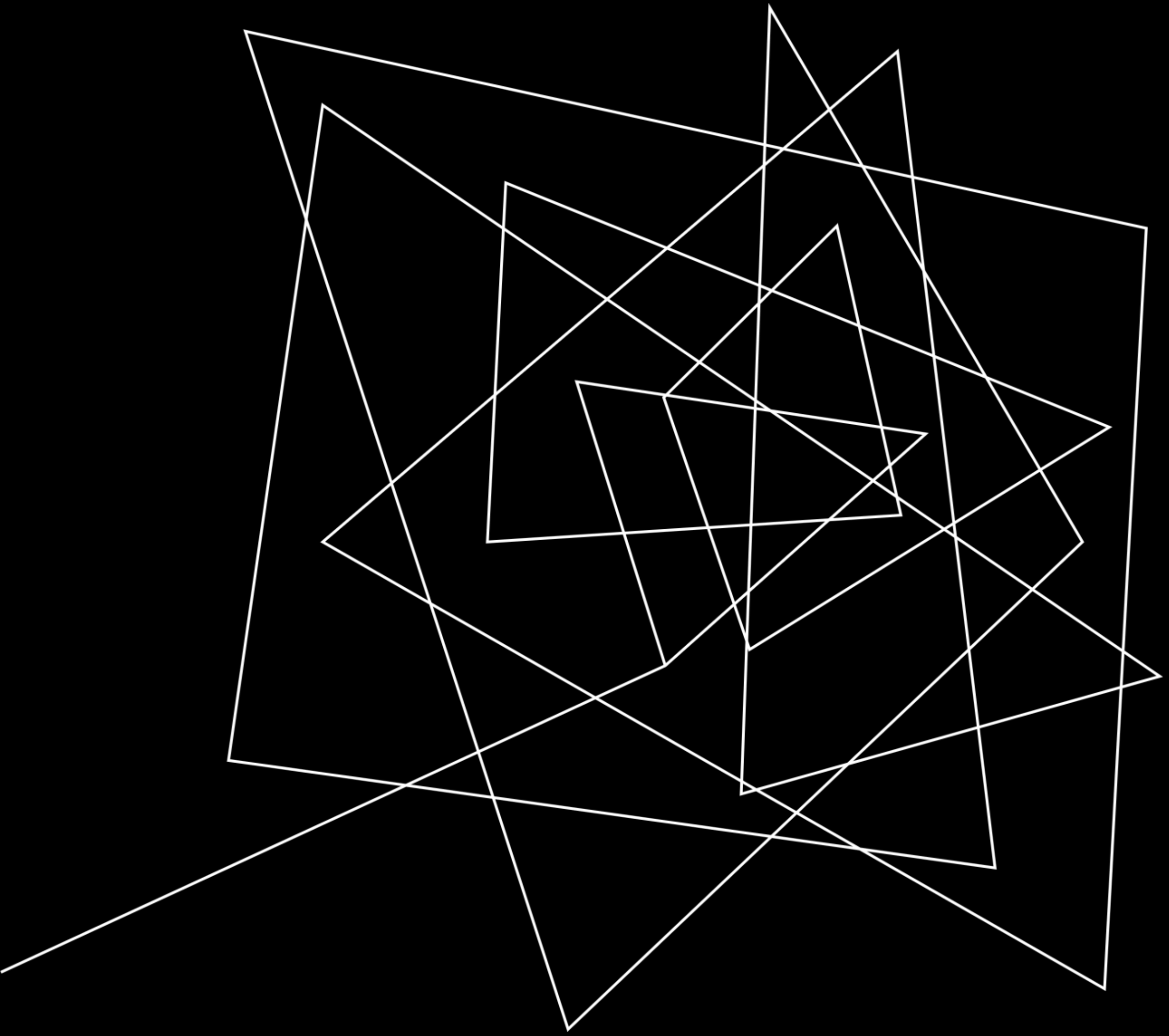
Structure of a request:

```
POST /v1/completions
Content-Type: application/json
{
  "model": "llama-3-8b-instruct",
  "prompt": "Explain quantization in simple terms.",
  "max_tokens": 100,
  "temperature": 0.7,
  "stream": false
}
```

Under the Hood: What Model Serving Actually Does

Structure of a response:

```
{
  "id": "cmpl-1c9b2e",
  "object": "text_completion",
  "created": 1730000000,
  "model": "llama-3-8b-instruct",
  "choices": [
    {
      "index": 0,
      "text": "Quantization means reducing the precision of the model’s numbers so it runs faster and uses less memory. For example, instead of storing weights as 32-bit floats, we can store them as 8-bit or 4-bit integers. This slightly reduces accuracy but greatly improves efficiency.",
      "finish_reason": "stop"
    }
  ]
}
```



Hardware and Optimization

Model Size Tiers & Typical Hardware

Tier	Params	Where It Runs Well	Notes
Tiny	<1B	Phones / NPUs / low-power iGPUs.	Great latency; short context.
Small	2–4B	8–12 GB VRAM GPUs.	Good latency; low reasoning.
Medium	6–8B	16–24 GB GPUs (e.g., 4070/4080/4090/5090).	Local sweet spot.
Upper-Mid	10–14B	24–48 GB GPUs (4090/A5000/RTX 6000 Ada).	Strong reasoning locally.
Large	20–40B	80 GB class or multi-GPU (A100/H100 RTX Pro 6000).	Consider tensor/pipeline parallel.
Massive	100–400B+	Unified-mem workstations (M3 Ultra/M4 128GB/ DGX Spark) or multi-node.	KV dominates; paging essential.

Quantization & Other Efficiency Magic

Quantization = compressing the model’s weights from high-precision (16-bit/32-bit) to lower bit formats (8-, 4-, even 2-bit).

Method / Trick	What It Does	Benefit	Trade-Off
8-bit Quantization	Compresses FP16 weights → INT8.	2× less memory.	Slight precision loss.
4-bit Quantization (AWQ, GPTQ, bitsandbytes)	Further compress weights.	Up to 70 % less VRAM.	Minor accuracy drop on rare tokens.
KV Cache Quantization	Stores attention cache in lower precision.	Keeps long chats in same VRAM.	Tiny drop in response fluency.
Offloading (CPU / Disk)	Move non-critical layers off GPU.	Run bigger models locally.	Slower first token latency.
LoRA Adapters (PEFT)	Train small low-rank weights instead of full model.	Fine-tune on laptop.	Limited update capacity.

Quantization & Other Efficiency Magic

- 🧠 Quantization = your best friend for 7B → 13B models on consumer GPUs (sub $400USD).
- 💾 Run 7B models in ~6 GB VRAM (4-bit).
- 🛠️ Tools: bitsandbytes, AWQ, AutoGPTQ, LMDeploy.
- ⚡ Combine with vLLM’s paged attention for huge speed ups.
- 🔋 Trade speed and efficiency wisely → find your sweet spot.

- FP16 = 100% VRAM.

- INT8 = 5 % VRAM.

- INT4 = 25% VRAM.

Understanding model requisites

What lives in memory during inference:

- Model weights (precision matters: FP16/INT8/INT4).
- KV cache for attention (grows with sequence length × batch).

KV cache size (per token).

For typical transformer decoders, an approachable form is:

KV bytes/token ≈ 2 × L × H × Dh × bytes_per_element.

Where L = layers, H = attention heads, Dh = head_dim (and H × Dh = hidden_size).

Simplified rule of thumb (since H×Dh = hidden_size d):

$$KV\ bytes/token\ \approx\ 2\ \times\ L\ \times\ d\ \times\ bytes\_per\_element.$$

$$Total\ memory\ \approx\ Weights\ +\ (KV\ bytes/token\ \times\ tokens\_in\_context\ \times\ batch\_size)$$

PagedAttention (vLLM) reduces KV **fragmentation**, letting you pack higher batches at the same memory.

Understanding model requisites

Worked examples (approx., FP16 KV)

- Llama-3-8B-class (L≈32, d≈4096):

  $KV \ \approx \ 2 \times 32 \times 4096 \times 2 \, bytes \ \approx \ 0.5 \, MB/token.$

  At 8k context, batch 4 → ~0.5 MB × 8,000 × 4 ≈ 16 GB KV alone (plus weights).

- Llama-2-13B-class (L≈40, d≈5120):

  $KV \ \approx \ 2 \times 40 \times 5120 \times 2 \ \approx \ 0.78 \, MB/token \ \rightarrow \ 8k \ ctx, batch \, 4 \ \approx \ 25 \, GB \ KV.$

  (Weight memory depends on quantization; e.g., INT4 weights ≈ 0.5 byte/param → 7B ≈ 3.5 GB; 13B ≈ 6.5 GB.).

What Actually Limits LLM Inference

- **Two: primary memory loads:**

  (1) Weights (precision/quantization) + **(2) KV cache** (context × batch).

- **Bandwidth > FLOPs** once the model fits: token decode is often memory-bound.

- **Paged KV (vLLM)** reduces fragmentation → bigger batches at same VRAM.

Latency, Throughput & Why Bandwidth Wins

Latency terms you must track (serving metrics):

- TTFT (Time-to-First-Token): initial latency before any token streams.
- TPOT (Time per Output Token): average gap between tokens once streaming.
- End-to-End latency: request submit → last token.
- Throughput (TPS): tokens/sec across all requests.

Why bandwidth rules:

Once the model fits, token decode is often memory-bound (moving KV + activations) more than FLOP-bound. Faster GDDR6/7 or unified memory bandwidth → higher TPS at the same precision. (See Mac Studio 819 GB/s, RTX 6000 Ada ~960 GB/s.)

What we really fit and how?

8–12 GB GPUs (Entry Local)

- 3–8B models in INT4/Q4 at comfortable context;
- 10–12B possible with trimmed context/throughput or CPU offload.

Community signal (r/LocalLLaMA/X[Formerly twitter]): many users report 7–8B working well on 8–12 GB with Q4, and pushing MoE/longer ctx with offload (lower TPS).

24 GB Class (RTX 3090 or 4090, 5090) (Local sweet spot)

- RTX 4090: 24 GB GDDR6X; great for 7–14B FP16 or ~20B INT4 with healthy context and batching.
- (FYI) 50-series introduces GDDR7 on some SKUs → more bandwidth; still check actual VRAM sizes per model/SKU. (Specs vary; plan capacity accordingly.)

What we really fit and how?

48 GB Workstations (RTX 6000 Ada / RTX Pro A5000-class)

- 10–20B possible with trimmed context/throughput or CPU offload for FP16 or upt to 40B quantized with good context.
- Multi-model (multiple smaller models, e.g. 8B models) for agentic workflow.

80 GB+ Workstations (RTX Pro 6000 /A100/ H100)

- 40B possible with trimmed context/throughput or CPU offload for FP16 or up to 70B quantized with good context.
- High inference speed.
- Complete agentic workflows.

What we really fit and how?

Unified-Memory (MacsM4 Max 128 GB; M3 Ultra up to 512 GB)

- M4 Max configs reach 128 GB unified memory (Mac Studio/MBP variants). Great for 70B with roomy context and batch.
- M3 Ultra Mac Studio: up to 512 GB unified memory and ~819 GB/s bandwidth. Confortable with 400B models with lengthy context.

Real-world reports

- Dave2D/press coverage: DeepSeek-R1 671B (Q4) loaded entirely in memory on M3 Ultra 512 GB, <200 W power; VRAM cap adjusted to ~448 GB. (Throughput modest but proves capacity.).
- LocalLLaMA / LocalLLM threads: users running 70–72B on 128 GB, and even ~123B at a few TPS depending on quantization and toolchain (llama.cpp/Ollama/MLX). Trade-off: slower TPS, shorter context, or both.

What we really fit and how?

CPU+RAM Inference (EPYC/Xeon) & Hybrid Overflow

- CPU-only (llama.cpp): works—bandwidth and NUMA tuning matter more than raw core count; modern EPYC shows solid tokens/s when memory channels are saturated.
- Hybrid overflow: keep weights on GPU (quantized), spill KV or part of weights to system RAM; cheaper capacity, slower throughput—but it lets you fit the model locally. Community reports also show DeepSeek R1 671B Q4 on 512 GB EPYC single-socket at a few TPS.

Real-world reports

- Modern CPUs can fit multiple TB of ram, your only constraint is TPS, models like Llama 70B at FP16 generate at about 3.2 TPS. Usable at small scale for single user, not serving.

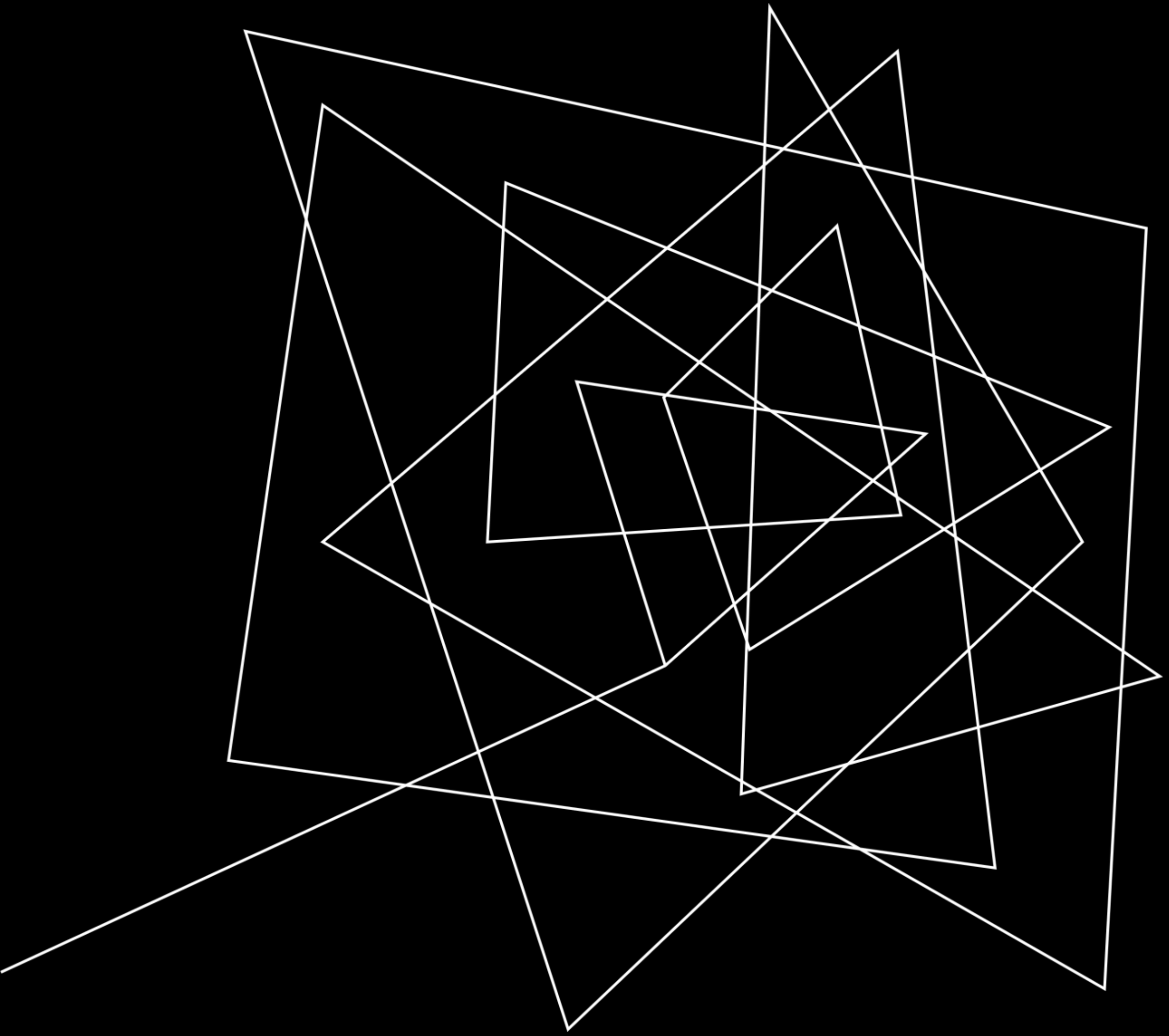
What we really fit and how?

Multi-GPU Setups (Nvidia DGX 640 GB, HGX B300 2.1 TB)

- 640 GB node: Comfortable up to ~200–250B params in BF16, or ~400–500B in FP8/INT8, assuming modest KV needs.
- B300 (~2.3 TB): Can host ~0.8–0.9T params in BF16 or ~1.6–1.8T in FP8/INT8—i.e., 400B-class BF16 becomes easy, and trillion-class becomes realistic in 8-bit. (These are capacity estimates; concrete batch/ctx changes the headroom.).

Real-world reports

- At this point, the idea paradigm is shifted. In modern large-scale deployments, companies focus on parameter counts to optimizing tokens processed per second. Once you operate with multi-GPU nodes like 8×H100s or B300s, capacity is no longer the bottleneck, throughput and latency define real performance. Enterprises care about how efficiently each watt and dollar turns into generated tokens, not just how many parameters fit in memory.



Front-Ends

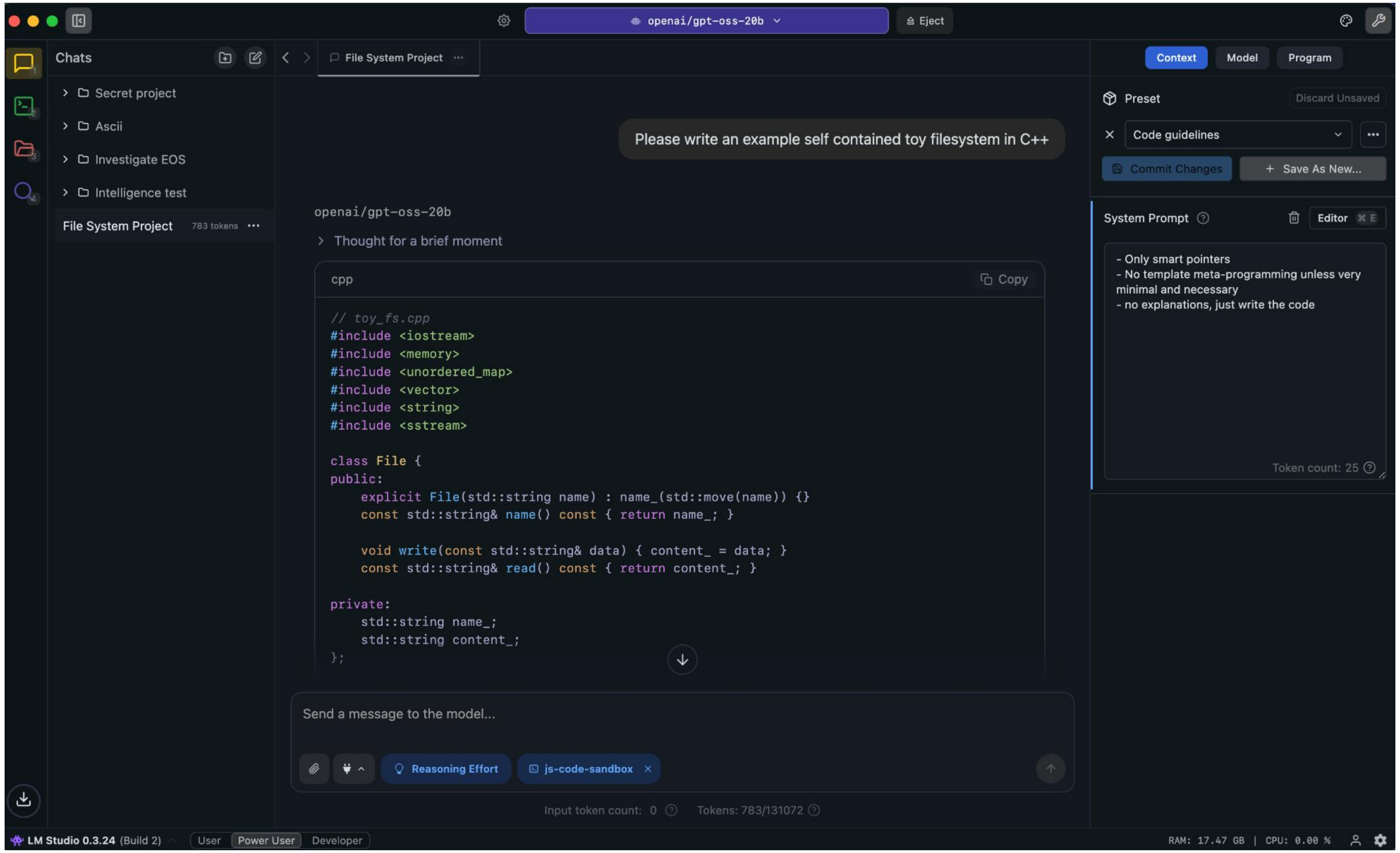
The Human Layer

- The front-end is how users (or downstream systems) interact with your model.
- It defines UX, latency perception, and data capture.
- Common front-end patterns:
  - Chat-style UIs (Open WebUI, LM Studio, Gradio, Streamlit).
  - API dashboards (Swagger, Postman).
  - Custom apps (React/Next.js, Electron, mobile).
- Key insight: Front-end doesn’t change the model — it changes how people experience it.

The Human Layer

Category	Tools	Notes
Local UIs	LM Studio, Open WebUI, Ollama GUI.	Simple, prebuilt chat interfaces.
Custom CLIs	llama.cpp, vLLM chat scripts.	For local workflows or automation.
Image Generation	Automatic1111 WebUI, ComfyUI, InvokeAI,, Diffusers WebUI.	Dedicated UIs for Stable Diffusion and other diffusion models.

LM Studio



Please write an example self contained toy filesystem in C++

openai/gpt-oss-20b

Thought for a brief moment

cpp

```cpp
// toy_fs.cpp
#include <iostream>
#include <memory>
#include <unordered_map>
#include <vector>
#include <string>
#include <sstream>

class File {
public:
    explicit File(std::string name) : name_(std::move(name)) {}
    const std::string& name() const { return name_; }

    void write(const std::string& data) { content_ = data; }
    const std::string& read() const { return content_; }

private:
    std::string name_;
    std::string content_;
};
```

Send a message to the model...

Input token count: 0

Tokens: 783/131072

Preset: Code guidelines

System Prompt

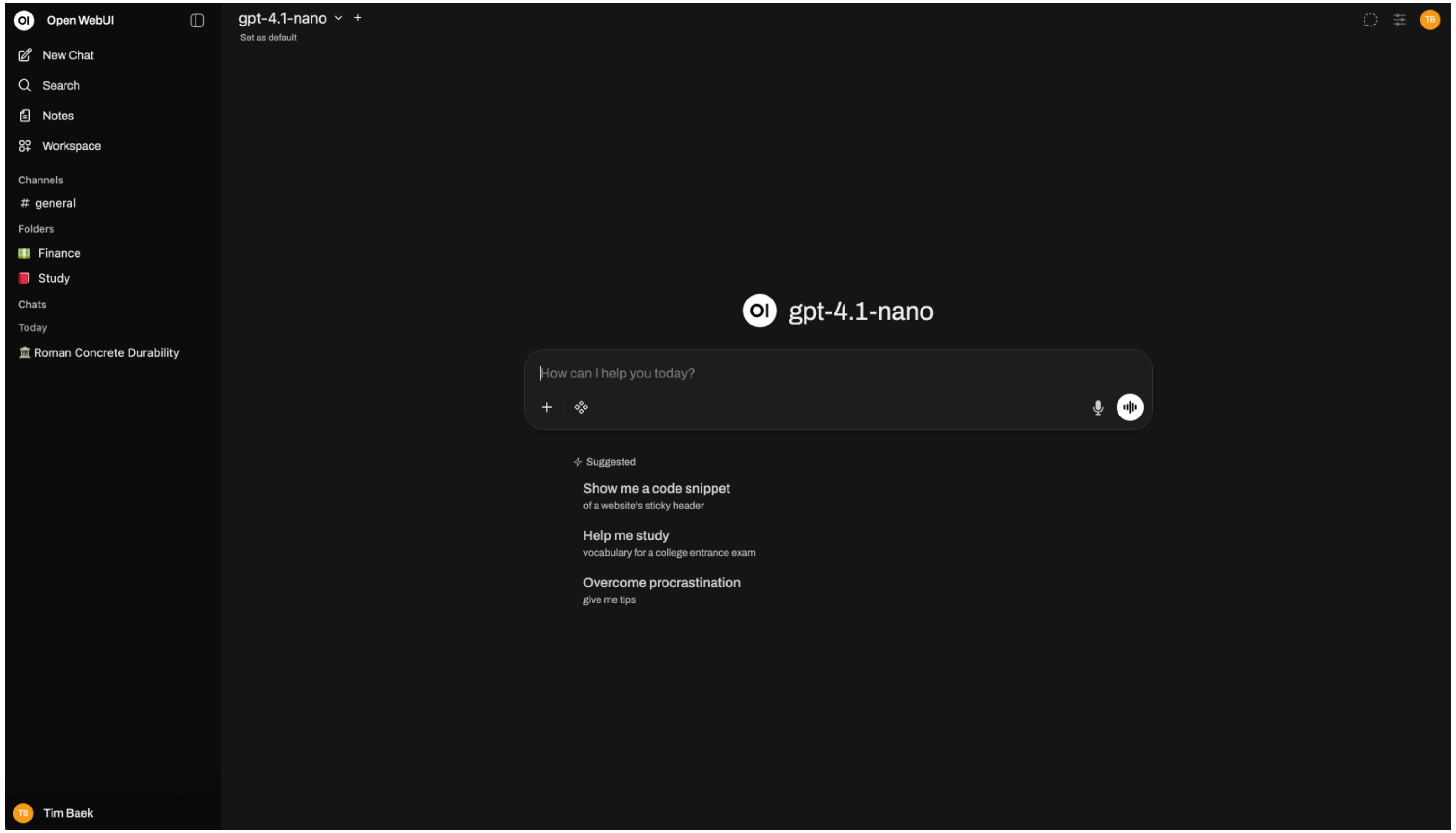
- Only smart pointers
- No template meta-programming unless very minimal and necessary
- no explanations, just write the code

Token count: 25

LM Studio 0.3.24 (Build 2) | User | Power User | Developer

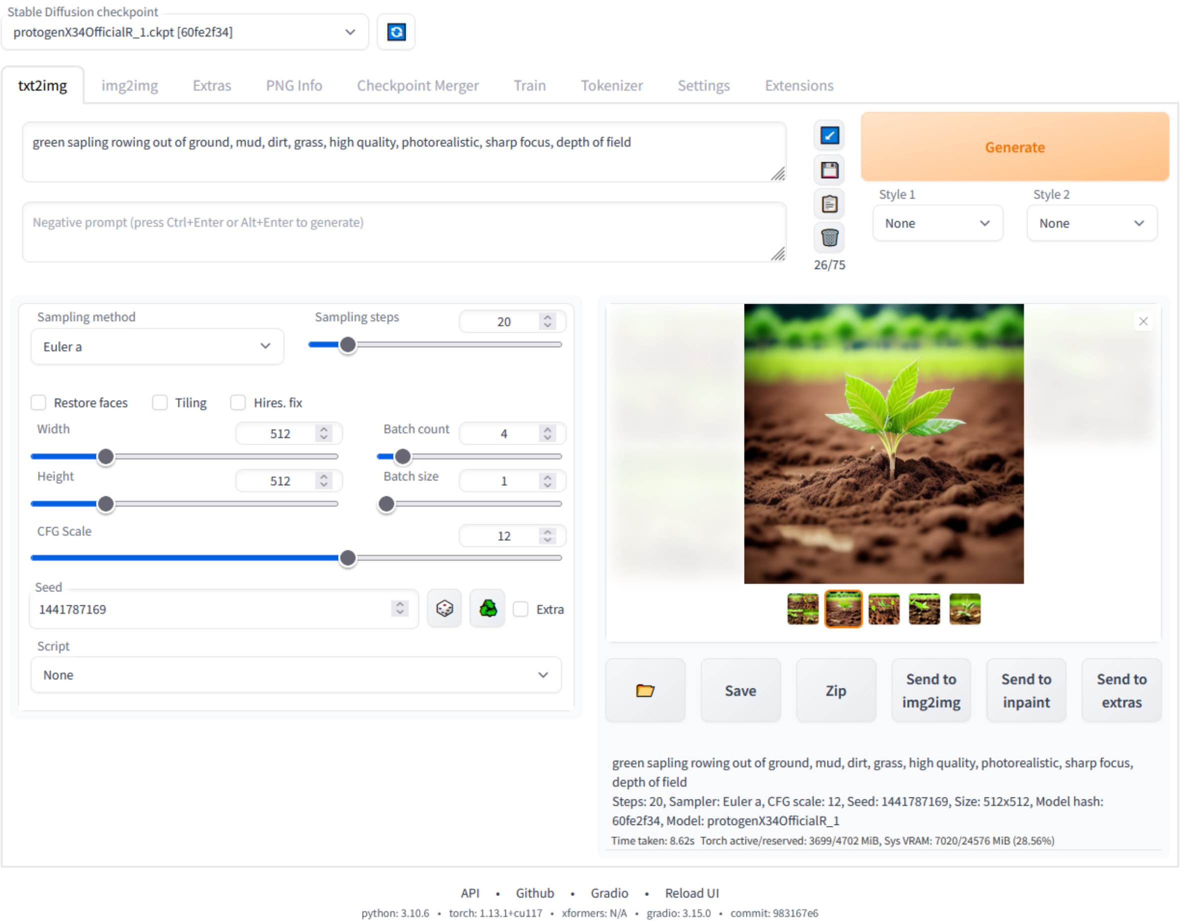
RAM: 17.47 GB | CPU: 0.00 %

Open WebUI



Open WebUI
New Chat
Search
Notes
Workspace
Channels
# general
Folders
Finance
Study
Chats
Today
Roman Concrete Durability
Tim Baek
gpt-4.1-nano
Set as default
gpt-4.1-nano
How can I help you today?
Suggested
Show me a code snippet
of a website's sticky header
Help me study
vocabulary for a college entrance exam
Overcome procrastination
give me tips

Automatic1111 WebUI



Stable Diffusion checkpoint: protogenX34OfficialR_1.ckpt [60fe2f34]

green sapling rowing out of ground, mud, dirt, grass, high quality, photorealistic, sharp focus, depth of field

Negative prompt (press Ctrl+Enter or Alt+Enter to generate)

26/75

Style 1: None

Style 2: None

Sampling method: Euler a

Sampling steps: 20

□ Restore faces □ Tiling □ Hires. fix

Width: 512

Height: 512

Batch count: 4

Batch size: 1

CFG Scale: 12

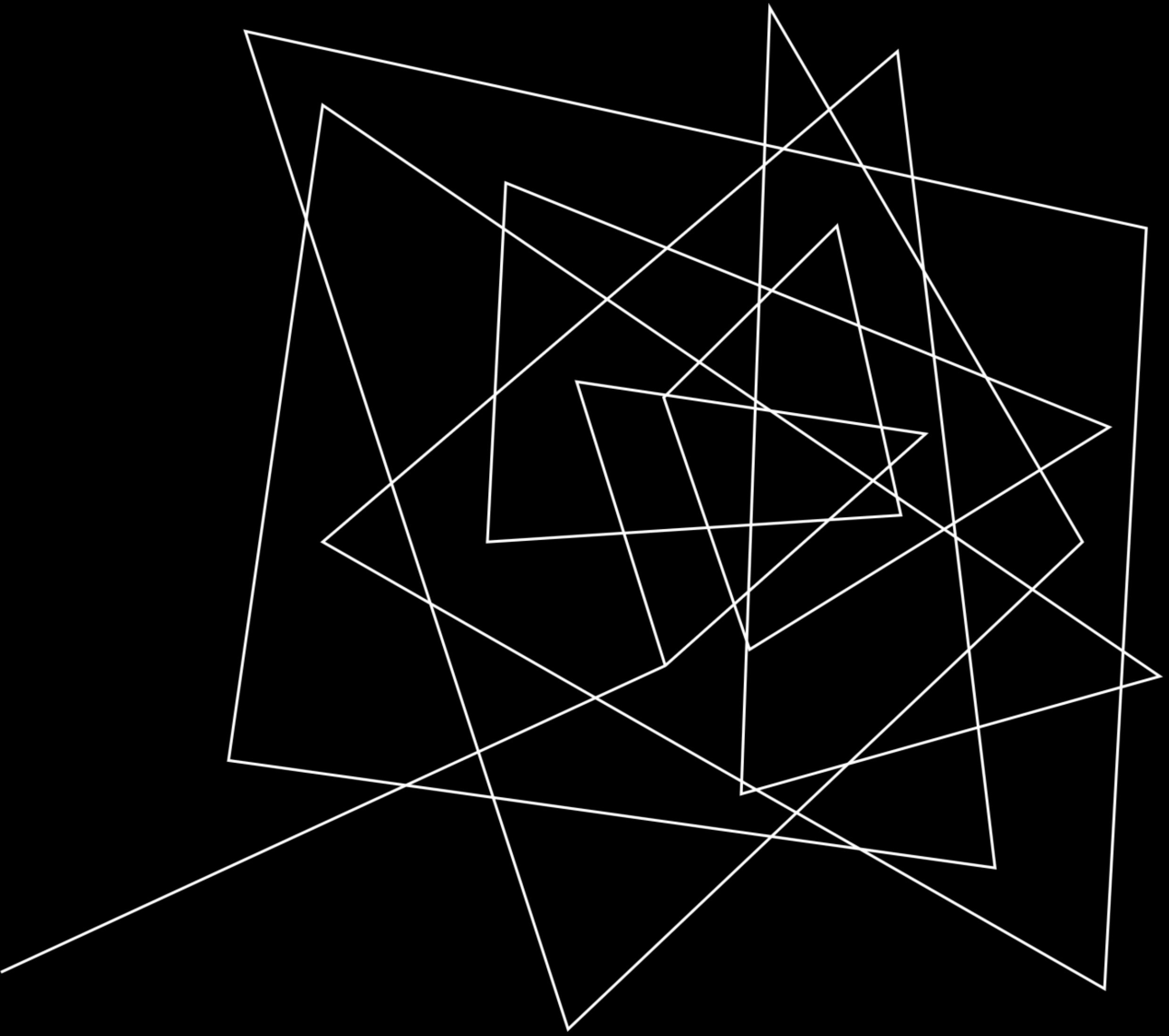
Seed: 1441787169 □ Extra

Script: None

green sapling rowing out of ground, mud, dirt, grass, high quality, photorealistic, sharp focus, depth of field
Steps: 20, Sampler: Euler a, CFG scale: 12, Seed: 1441787169, Size: 512x512, Model hash: 60fe2f34, Model: protogenX34OfficialR_1
Time taken: 8.62s Torch active/reserved: 3699/4702 MiB, Sys VRAM: 7020/24576 MiB (28.56%)

API • Github • Gradio • Reload UI

python: 3.10.6 • torch: 1.13.1+cu117 • xformers: N/A • gradio: 3.15.0 • commit: 983167e6



Integration and Apis

Integration & APIs

How front-ends talk to models.

Two major routes:

1. OpenAI-compatible REST endpoints (vLLM, Ollama, LM Studio)
2. Direct inference pipelines (Transformers API, llama.cpp C++ bindings)

Most open serving engines expose `/v1/completions` or `/v1/chat/completions`.

That’s your universal contract.

Integration & APIs

Integrate into your own app:

Local API → App → Backend → User.

Typical stack:

- Frontend: React, Vue, Swift, Android.
- Backend: FastAPI / Express.
- Inference: vLLM / Ollama / LM Studio.

Libraries:

LangChain, LlamaIndex, Transformers, OpenAI SDK.

Integration & APIs

[User Interface]

↓

[API Gateway / Router]

↓

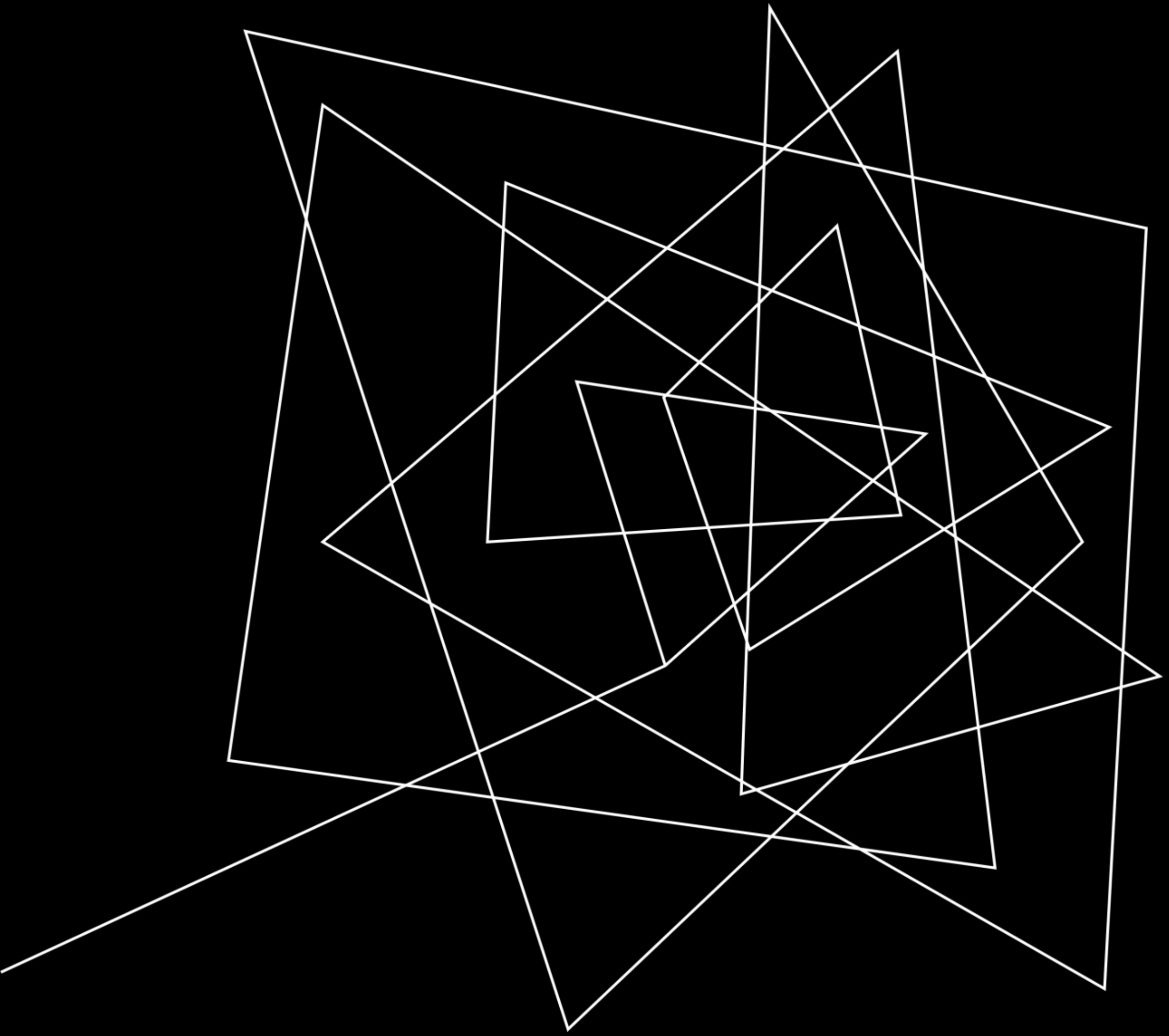
[Model Serving (vLLM / Ollama)]

↓

[GPU / CPU Hardware]

↓

[Logs / Monitoring / Storage]



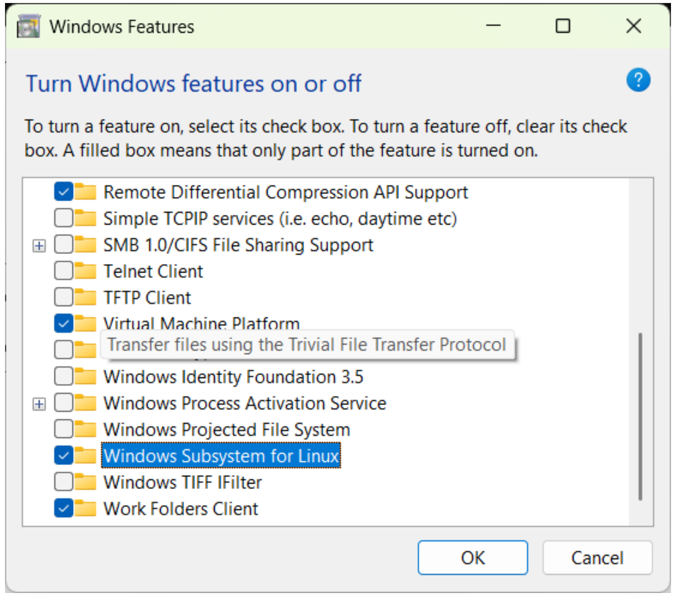
Full Hands-On Workflow

WSL

- Most of this backends are designed to work with linux, since no one in their right mind would run a windows server.
- WSL (Windows Subsystems for Linux) lets us run a full Linux environment inside windows. Provides a Linux-compatible kernel layer that translates Linux system calls into Windows system calls.
- If you have a Linux partition you can go and boot to that and skip this steps.

Installing WSL

- Inside windows powershell run `wsl --install`
- Activate WSL inside Windows features.



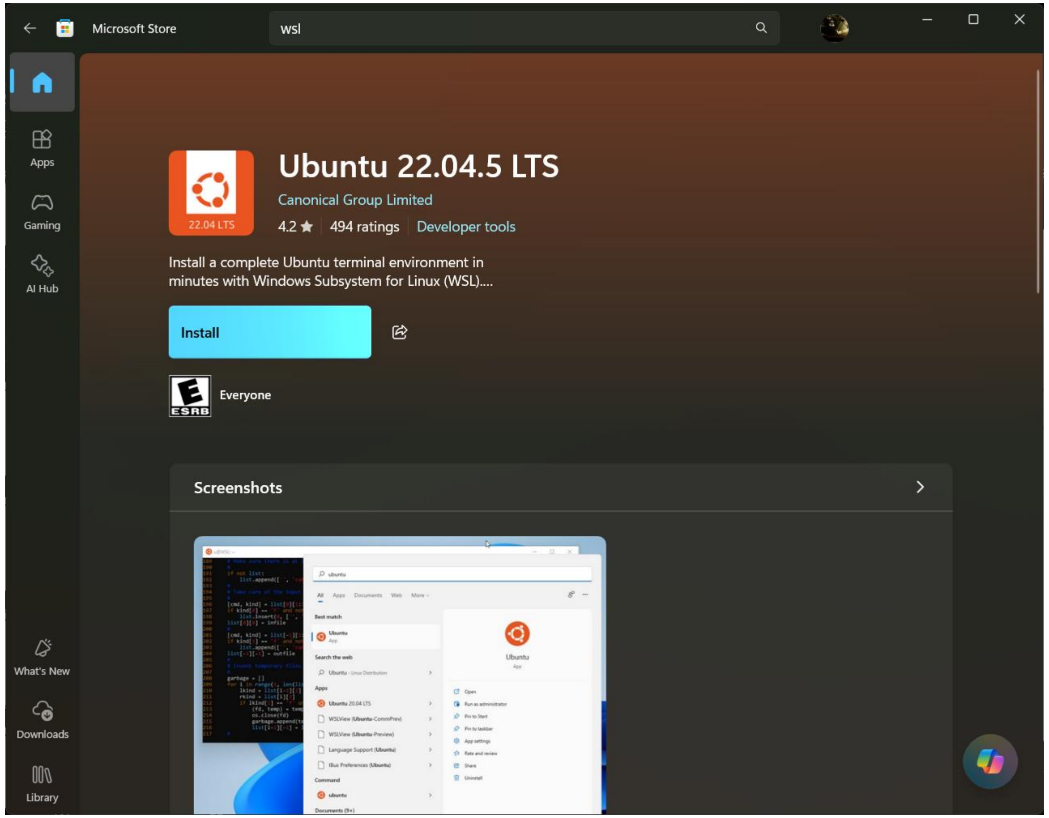
Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- ☑ Remote Differential Compression API Support
- ☐ Simple TCPIP services (i.e. echo, daytime etc)
- ☐ SMB 1.0/CIFS File Sharing Support
- ☐ Telnet Client
- ☐ TFTP Client
- ☑ Virtual Machine Platform
- ☐ Transfer files using the Trivial File Transfer Protocol
- ☐ Windows Identity Foundation 3.5
- ☐ Windows Process Activation Service
- ☐ Windows Projected File System
- ☑ Windows Subsystem for Linux
- ☐ Windows TIFF IFilter
- ☑ Work Folders Client

Installing WSL

- Install Ubuntu from windows store



Ubuntu 22.04.5 LTS

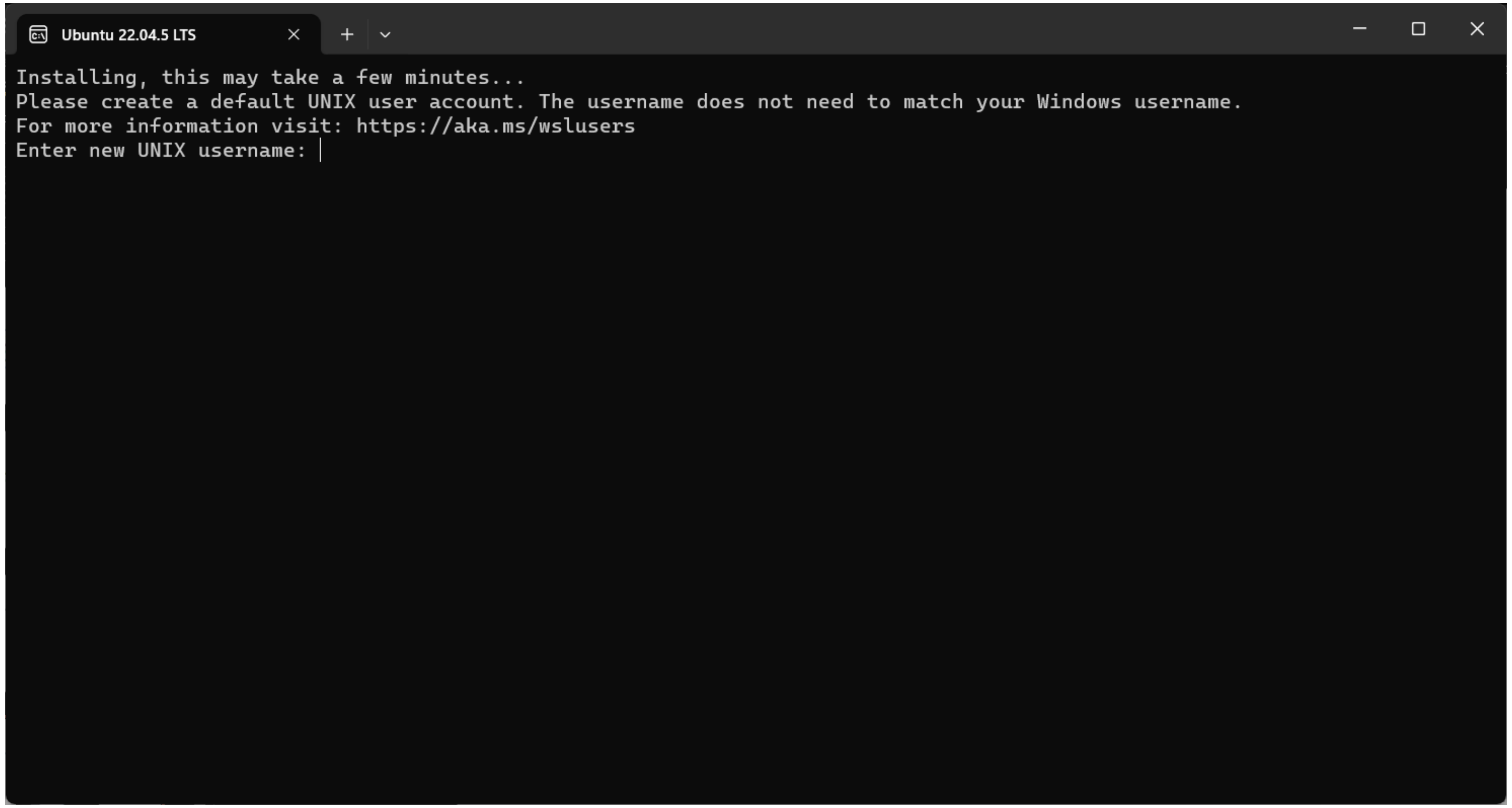
Canonical Group Limited

4.2 ★ | 494 ratings | Developer tools

Install a complete Ubuntu terminal environment in minutes with Windows Subsystem for Linux (WSL)....

Installing WSL

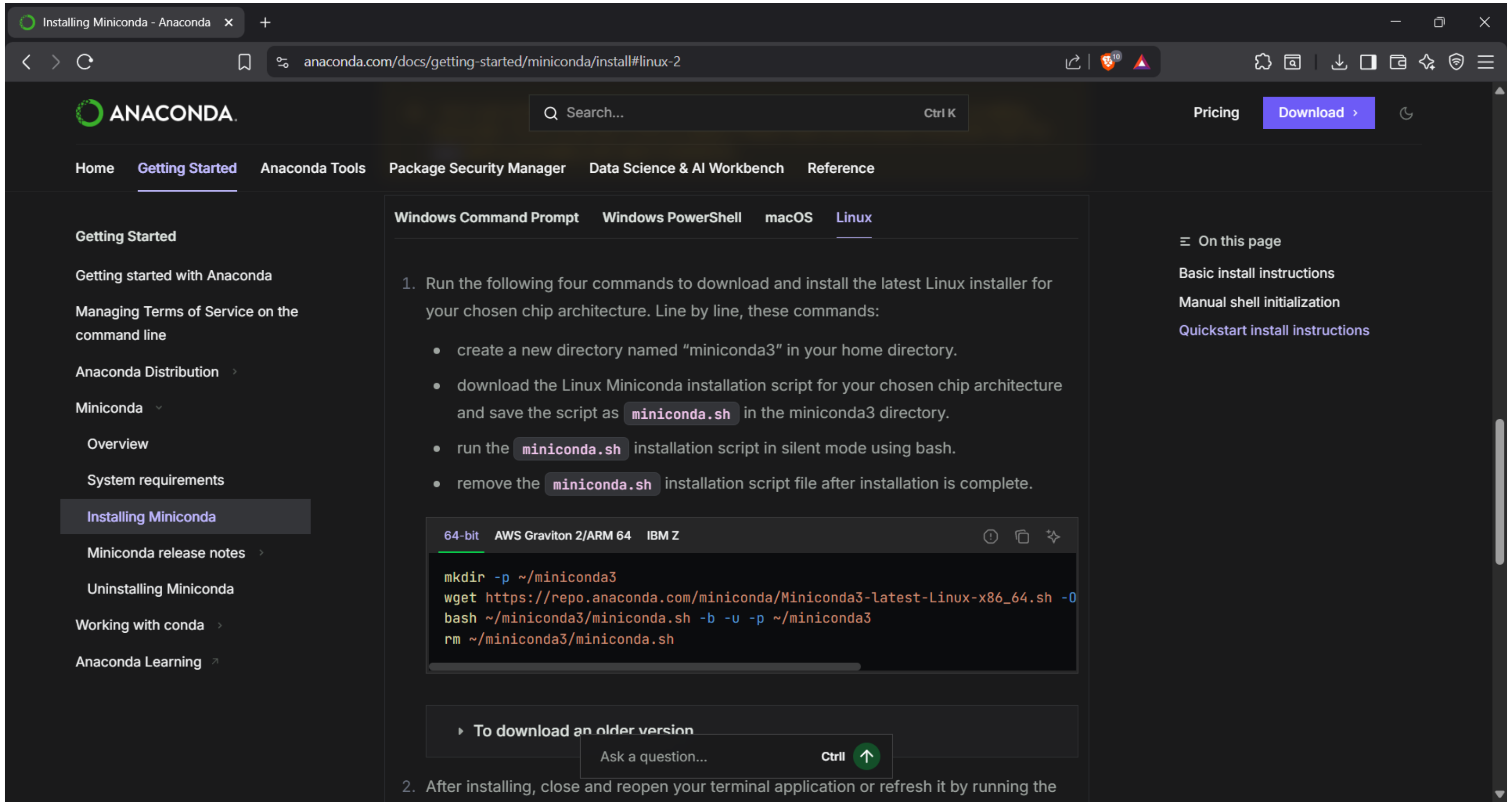
- Follow screen prompts to configure ubuntu.



```
Installing, this may take a few minutes...
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username:
```

Installing Miniconda

- Install miniconda following scripts from page.

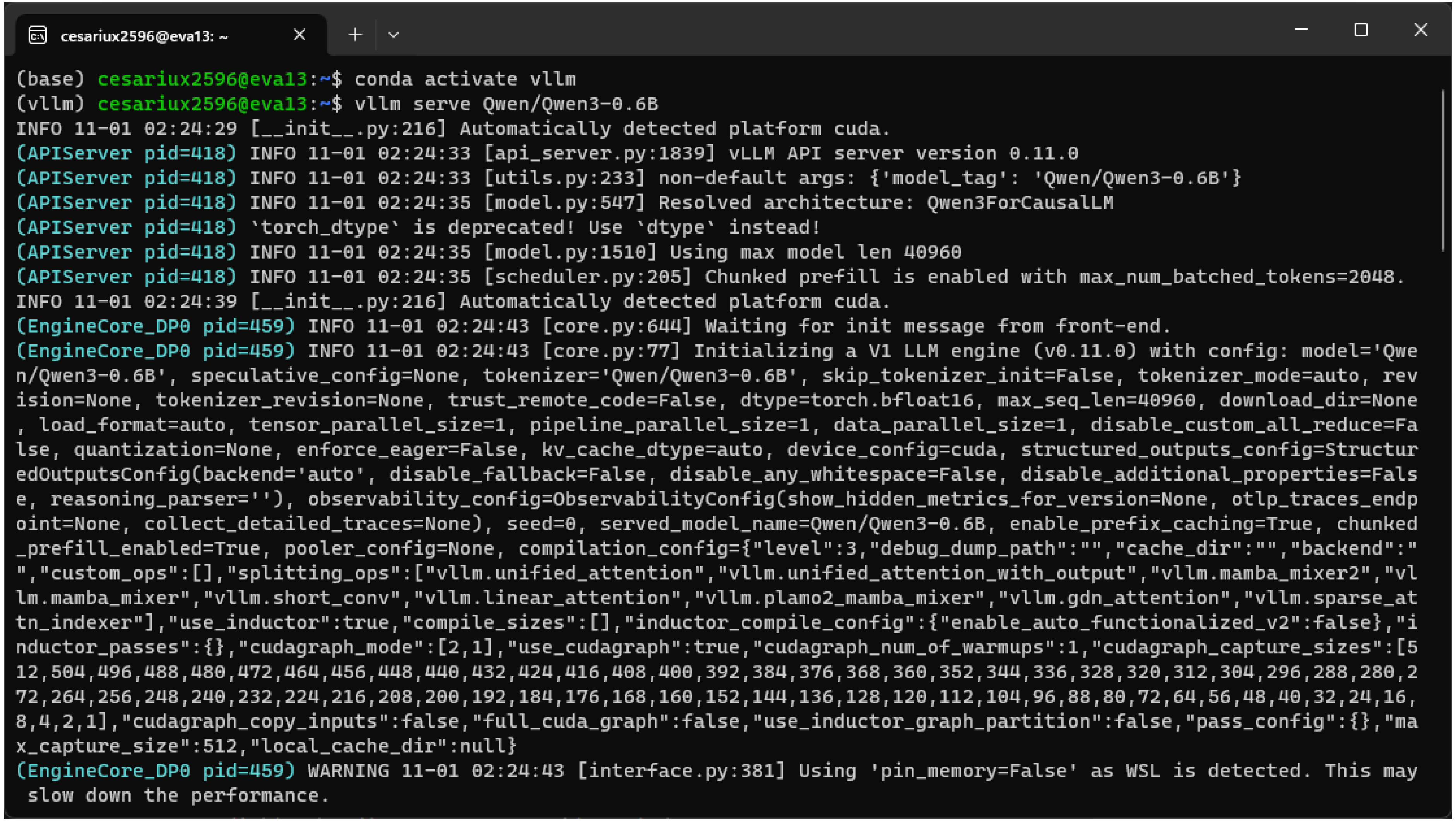


1. Run the following four commands to download and install the latest Linux installer for your chosen chip architecture. Line by line, these commands:
   - create a new directory named “miniconda3” in your home directory.
   - download the Linux Miniconda installation script for your chosen chip architecture and save the script as `miniconda.sh` in the miniconda3 directory.
   - run the `miniconda.sh` installation script in silent mode using bash.
   - remove the `miniconda.sh` installation script file after installation is complete.

```
mkdir -p ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm ~/miniconda3/miniconda.sh
```

2. After installing, close and reopen your terminal application or refresh it by running the

vLLM



```
(base) cesariux2596@eva13:~$ conda activate vllm
(vllm) cesariux2596@eva13:~$ vllm serve Qwen/Qwen3-0.6B
INFO 11-01 02:24:29 [__init__.py:216] Automatically detected platform cuda.
(APIServer pid=418) INFO 11-01 02:24:33 [api_server.py:1839] vLLM API server version 0.11.0
(APIServer pid=418) INFO 11-01 02:24:33 [utils.py:233] non-default args: {'model_tag': 'Qwen/Qwen3-0.6B'}
(APIServer pid=418) INFO 11-01 02:24:35 [model.py:547] Resolved architecture: Qwen3ForCausalLM
(APIServer pid=418) `torch_dtype` is deprecated! Use `dtype` instead!
(APIServer pid=418) INFO 11-01 02:24:35 [model.py:1510] Using max model len 40960
(APIServer pid=418) INFO 11-01 02:24:35 [scheduler.py:205] Chunked prefill is enabled with max_num_batched_tokens=2048.
INFO 11-01 02:24:39 [__init__.py:216] Automatically detected platform cuda.
(EngineCore_DP0 pid=459) INFO 11-01 02:24:43 [core.py:644] Waiting for init message from front-end.
(EngineCore_DP0 pid=459) INFO 11-01 02:24:43 [core.py:77] Initializing a V1 LLM engine (v0.11.0) with config: model='Qwe
n/Qwen3-0.6B', speculative_config=None, tokenizer='Qwen/Qwen3-0.6B', skip_tokenizer_init=False, tokenizer_mode=auto, rev
ision=None, tokenizer_revision=None, trust_remote_code=False, dtype=torch.bfloat16, max_seq_len=40960, download_dir=None
, load_format=auto, tensor_parallel_size=1, pipeline_parallel_size=1, data_parallel_size=1, disable_custom_all_reduce=Fa
lse, quantization=None, enforce_eager=False, kv_cache_dtype=auto, device_config=cuda, structured_outputs_config=Structur
edOutputsConfig(backend='auto', disable_fallback=False, disable_any_whitespace=False, disable_additional_properties=Fals
e, reasoning_parser=''), observability_config=ObservabilityConfig(show_hidden_metrics_for_version=None, otlp_traces_endp
oint=None, collect_detailed_traces=None), seed=0, served_model_name=Qwen/Qwen3-0.6B, enable_prefix_caching=True, chunked
_prefill_enabled=True, pooler_config=None, compilation_config={"level":3,"debug_dump_path":"","cache_dir":"","backend":"
","custom_ops":[],"splitting_ops":["vllm.unified_attention","vllm.unified_attention_with_output","vllm.mamba_mixer2","vl
lm.mamba_mixer","vllm.short_conv","vllm.linear_attention","vllm.plamo2_mamba_mixer","vllm.gdn_attention","vllm.sparse_at
tn_indexer"],"use_inductor":true,"compile_sizes":[],"inductor_compile_config":{"enable_auto_functionalized_v2":false},"i
nductor_passes":{},"cudagraph_mode":[2,1],"use_cudagraph":true,"cudagraph_num_of_warmups":1,"cudagraph_capture_sizes":[5
12,504,496,488,480,472,464,456,448,440,432,424,416,408,400,392,384,376,368,360,352,344,336,328,320,312,304,296,288,280,2
72,264,256,248,240,232,224,216,208,200,192,184,176,168,160,152,144,136,128,120,112,104,96,88,80,72,64,56,48,40,32,24,16,
8,4,2,1],"cudagraph_copy_inputs":false,"full_cuda_graph":false,"use_inductor_graph_partition":false,"pass_config":{},"ma
x_capture_size":512,"local_cache_dir":null}
(EngineCore_DP0 pid=459) WARNING 11-01 02:24:43 [interface.py:381] Using 'pin_memory=False' as WSL is detected. This may
 slow down the performance.
```

Install vLLM

1. Open terminal (Linux) or WSL (Windows)

   Apple silicon still in beta, must be build from scratch, follow installation guide:

   https://docs.vllm.ai/en/latest/getting_started/installation/cpu.html

2. Create a new environment for the dependencies

   ```
   conda  create -n vllm python=3.12 -y
   ```

3. Install the package

   ```
   pip install vllm
   ```

4. Serve a model to start the service

   ```
   vllm serve Qwen/Qwen3-0.6B
   ```

Why vLLM?

Aspect	Pros ✅	Cons ⚠️
Ease of Use	Straightforward CLI and Docker support; OpenAI-compatible API	Requires Python setup, CUDA drivers, and correct model configuration
Model Library	Loads any Hugging Face model (Llama 3, Mistral, Gemma, Qwen, etc.)	No built-in model browser or GUI
API Compatibility	Fully OpenAI-style `/v1/completions`, `/v1/chat/completions`, `/v1/embeddings`	Some endpoints still evolving; limited docs for advanced features
Performance	Industry-leading throughput via *PagedAttention*, continuous batching, FP8/INT8 support	High VRAM requirement; tuning needed for best performance
Privacy	100 % local inference, self-hosted on your own hardware or cloud	No native user auth or isolation — must add externally
Deployment	Scalable backend, integrates easily with LangChain, FastAPI, or custom apps	More complex scaling (multi-GPU or multi-node setup requires config)

vLLM API

- POST `/v1/completions` — single-turn text generation (like /completions).
- POST `/v1/chat/completions`— multi-turn chat endpoint, keeps conversation history.
- POST `/v1/embeddings` — embeddings generation endpoint.
- POST `/v1/models` — lists all models currently available locally.

Serving Models with vLLM

vLLM provides ways to serve modesl via API and interactive mode.

```
vllm serve Qwen/Qwen3-0.6B --dtype auto --api-key token-abc123
```

This loads the model and exposes the OpenAI-style HTTP API on a default port (e.g., 8000) for apps or front-ends.

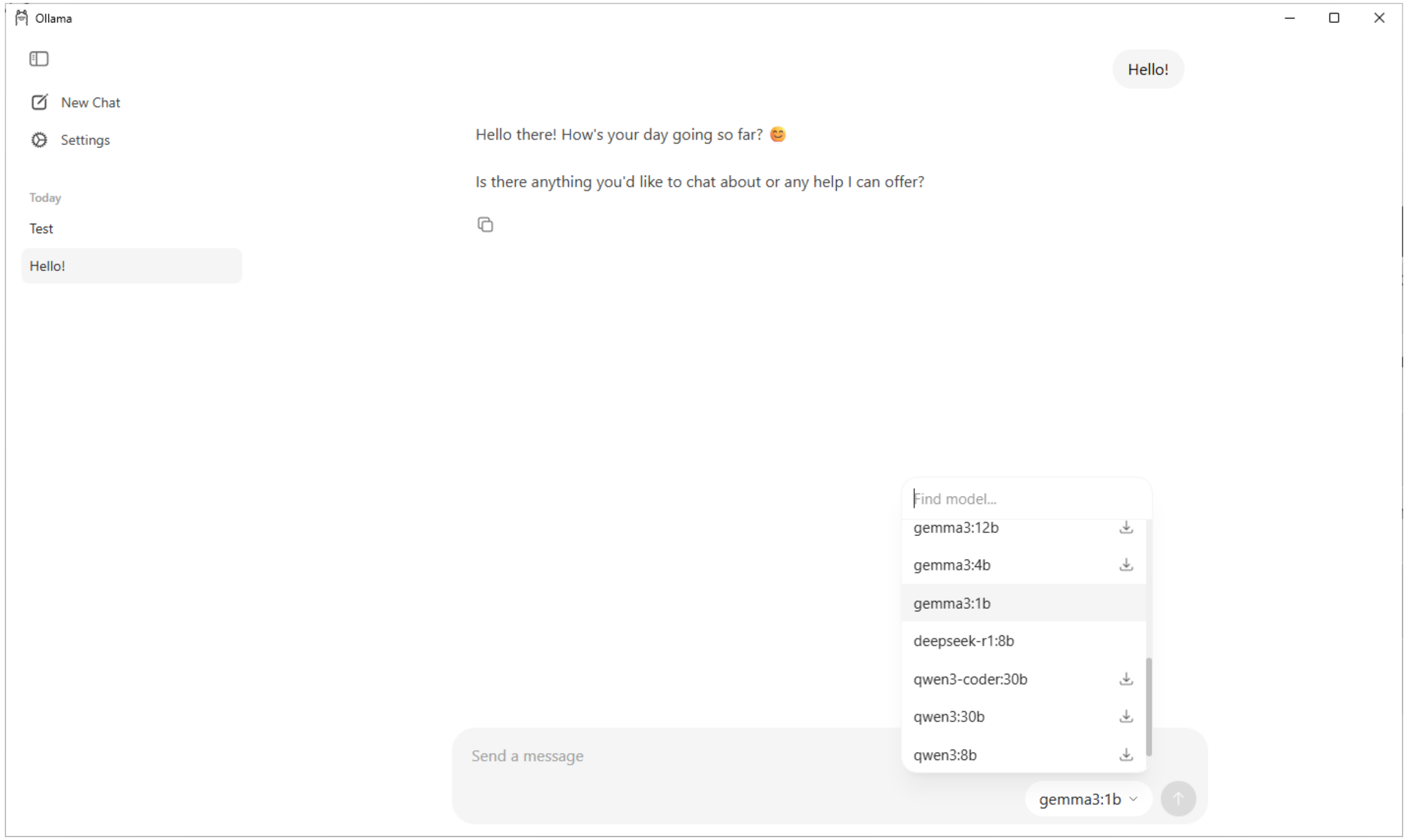
```
from vllm import LLM, SamplingParams

llm = LLM(model="facebook/opt-125m")

outputs = llm.generate(["Hello, how are you?"], SamplingParams(temperature=0.8, top_p=0.95))
```

This uses the Python LLM interface for direct inference outside the HTTP server

Ollama



Hello!

Hello there! How's your day going so far? 😊

Is there anything you'd like to chat about or any help I can offer?

Install Ollama

1. Go to https://ollama.com/download
2. Select your respective OS (macOS, Linux or Windows)
3. Follow Installation instructions (terminal for linux, wizard for Windows)
4. Download a model from the list

Why Ollama?

Aspect	Pros ✅	Cons ⚠️
Ease of Use	One-line setup, automatic GPU detection, beginner-friendly	Limited low-level tuning or GPU control
Model Library	Large curated catalog (Llama 3, Mistral, Qwen, Gemma, etc.)	No training or adapter merging support
API Compatibility	OpenAI-style REST API, works with LangChain, Postman, etc.	Basic concurrency; lacks enterprise features
Performance	Optimized GGUF quantized models, runs on CPU/GPU	Not ideal for heavy multi-user workloads
Privacy	100 % local inference — data never leaves your system	No built-in auth or multi-user management
Deployment	Cross-platform, Docker-ready, easy LAN hosting	Manual setup needed for scaling or load-balancing

Ollama API

- POST`/api/generate` — single-turn text generation (like /completions).
- POST `/api/chat` — multi-turn chat endpoint, keeps conversation history.
- GET `/api/tags` — lists all models currently available locally.
- POST `/api/pull` — downloads a model from the Ollama model library.
- POST `/api/create` — builds a custom model from a Modelfile (domain tuning).
- DELETE `/api/delete` — removes a model from local storage.

Serving Models with Ollama

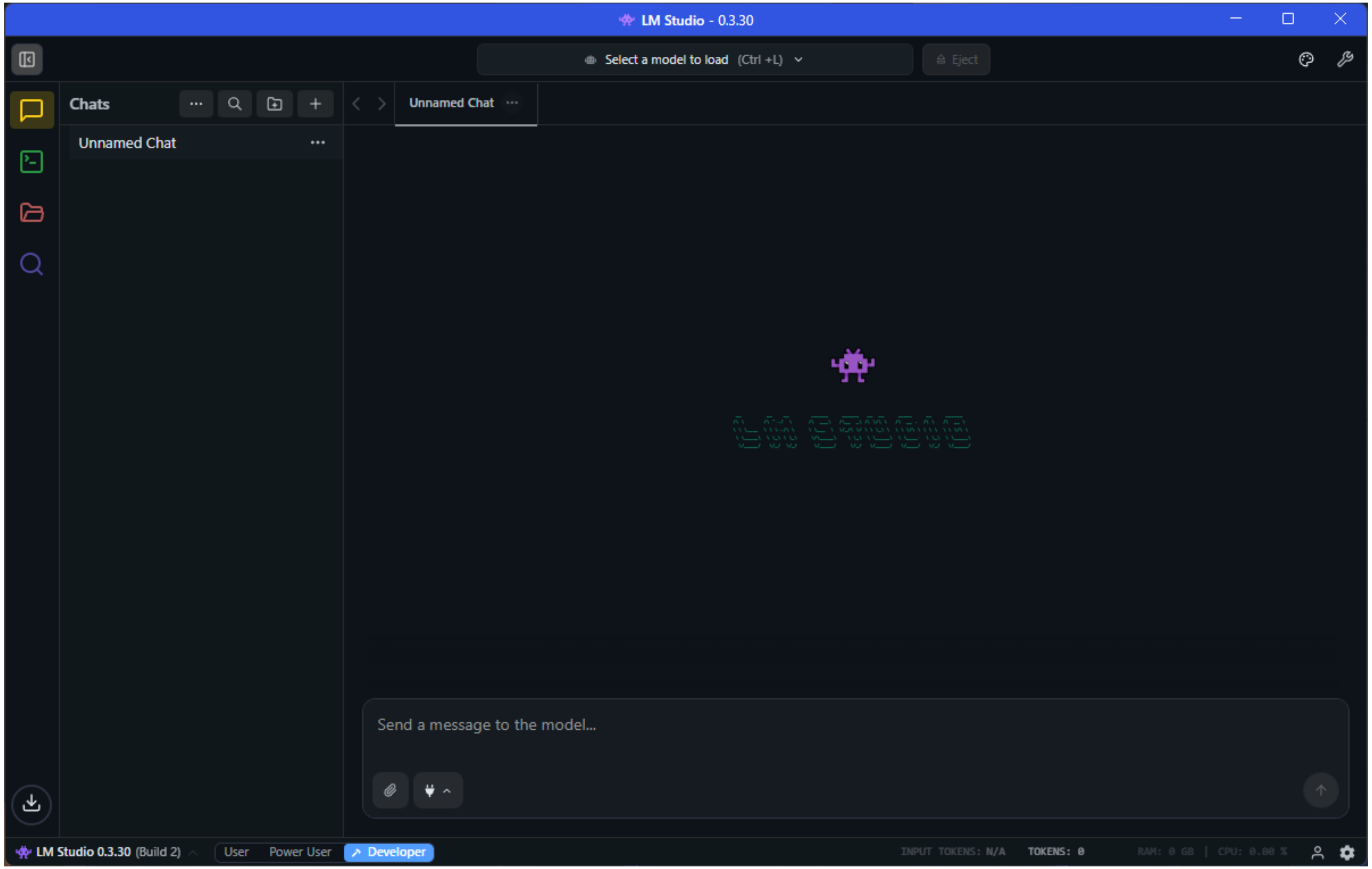
Ollama provides two simple commands to run your models locally or as a backend.

You can use either ollama serve or ollama run to start interacting with a model.

- `ollama serve` launches the local API server, ideal for connecting through Postman, LangChain, or your own app.
- `ollama run <model>` directly starts a model session for quick tests or CLI chats.

Once the server is running, any connected client can generate, chat, or list models using Ollama’s API.

LM Studio


LM Studio - 0.3.30
Select a model to load (Ctrl +L)
Eject
Chats
Unnamed Chat
Unnamed Chat
Send a message to the model...
LM Studio 0.3.30 (Build 2)
User
Power User
Developer
INPUT TOKENS: N/A
TOKENS: 0
RAM: 0 GB | CPU: 0.00 %

Install LM Studio

1. Go to https://lmstudio.ai/download.
2. Select your respective platform (macOS, Linux or Windows).
3. Follow Installation instructions (terminal for linux, wizard for Windows).
4. Download a model from the list.
5. Start chatting.

Why LM studio?

Aspect	Pros ✅	Cons ⚠️
Ease of Use	Extremely user-friendly GUI; one-click model downloads and local chat	Headless/server mode setup requires manual enabling
Model Library	Integrated catalog with local GGUF and Hugging Face model support	Smaller curated list than Ollama; no built-in fine-tuning
API Compatibility	Full OpenAI-style REST API	Some SDK and endpoint features still under development
Performance	Efficient llama.cpp backend with GPU/CPU acceleration and quantization	Limited advanced tuning (batch size, KV-cache control)
Privacy	100 % local inference; runs fully offline	No native user authentication or role management
Deployment	Cross-platform (Windows, macOS, Linux); supports GUI and headless server mode	Not ideal for multi-instance or large-scale serving

LM Studio API

- POST `/v1/completions` — single-turn text generation (like /completions).
- POST `/v1/chat/completions`— multi-turn chat endpoint, keeps conversation history.
- POST `/v1/embeddings` — embeddings generation endpoint.
- POST `/v1/models` — lists all models currently available locally.

Serving Models with LM Studio

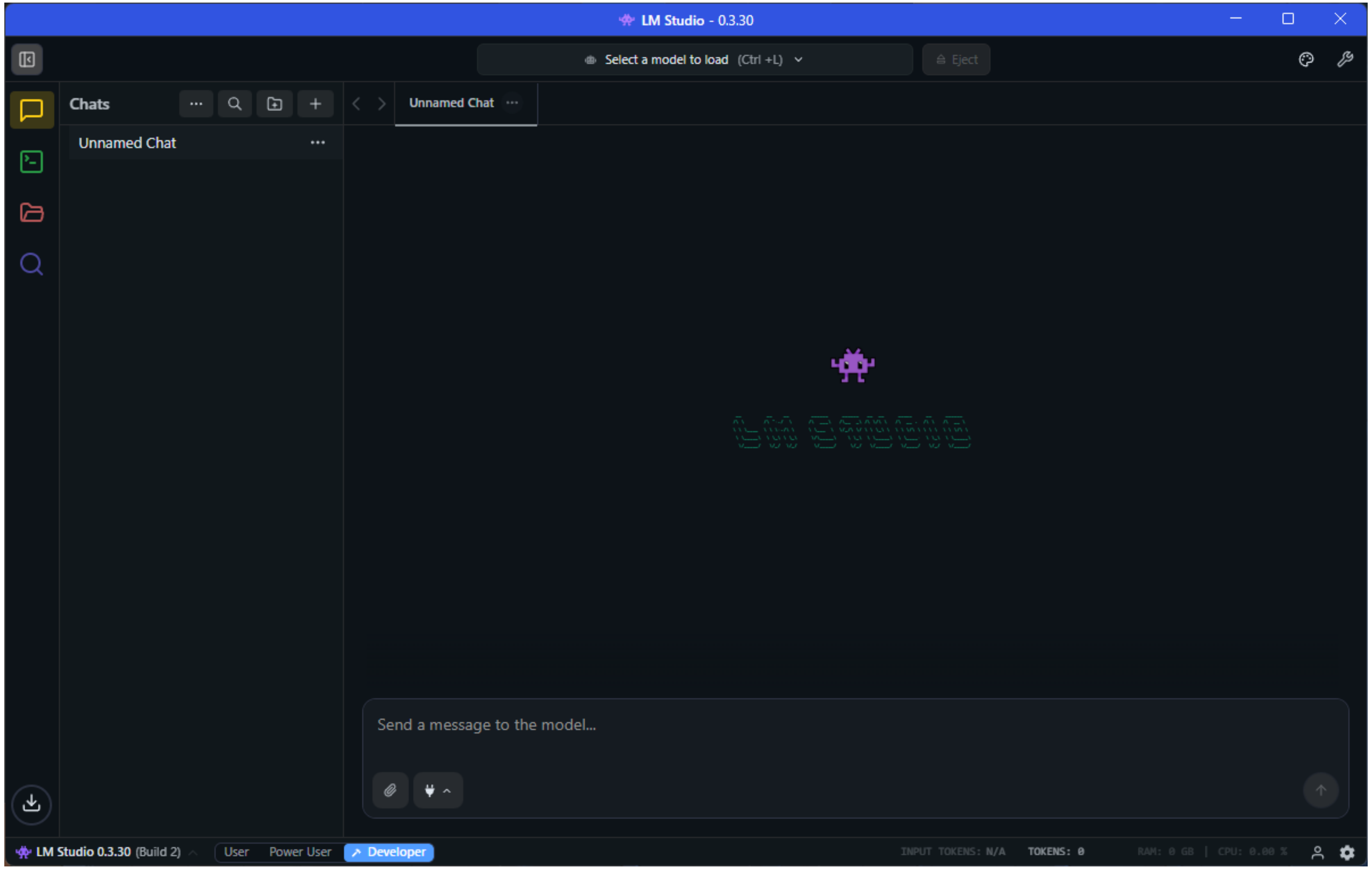
LM Studio can be run as a service without the GUI. This is useful for running LM Studio on a server or in the background on your local machine. This works on Mac, Windows, and Linux machines with a graphical user interface.

To enable this, head to app settings (Cmd / Ctrl + ,) and check the box to run the LLM server on login.

To start the service use: `Lms server start`

Your latest server state will be restored.

LM Studio


LM Studio - 0.3.30
Select a model to load (Ctrl +L)
Eject
Chats
Unnamed Chat
Unnamed Chat
Send a message to the model...
LM Studio 0.3.30 (Build 2)
User
Power User
Developer
INPUT TOKENS: N/A
TOKENS: 0
RAM: 0 GB | CPU: 0.00 %

Install LM Studio

1. Go to https://lmstudio.ai/download
2. Select your respective platform (macOS, Linux or Windows)
3. Follow Installation instructions (terminal for linux, wizard for Windows)
4. Download a model from the list

Why LM studio?

Aspect	Pros ✅	Cons ⚠️
Ease of Use	Extremely user-friendly GUI; one-click model downloads and local chat	Headless/server mode setup requires manual enabling
Model Library	Integrated catalog with local GGUF and Hugging Face model support	Smaller curated list than Ollama; no built-in fine-tuning
API Compatibility	Full OpenAI-style REST API	Some SDK and endpoint features still under development
Performance	Efficient llama.cpp backend with GPU/CPU acceleration and quantization	Limited advanced tuning (batch size, KV-cache control)
Privacy	100 % local inference; runs fully offline	No native user authentication or role management
Deployment	Cross-platform (Windows, macOS, Linux); supports GUI and headless server mode	Not ideal for multi-instance or large-scale serving

LM Studio API

- POST `/v1/completions` — single-turn text generation (like /completions).
- POST `/v1/chat/completions`— multi-turn chat endpoint, keeps conversation history.
- POST `/v1/embeddings` — embeddings generation endpoint.
- POST `/v1/models` — lists all models currently available locally.

Serving Models with LM Studio

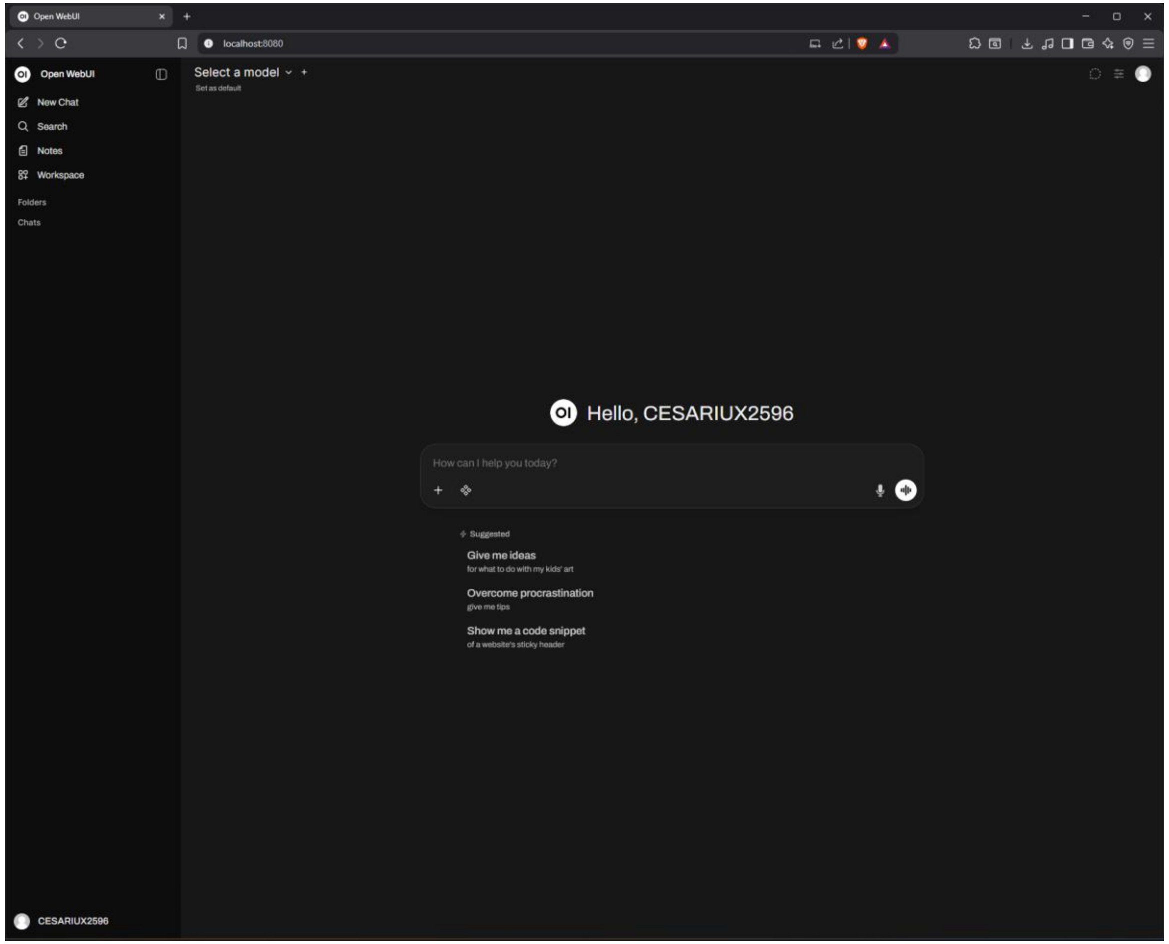
LM Studio can be run as a service without the GUI. This is useful for running LM Studio on a server or in the background on your local machine. This works on Mac, Windows, and Linux machines with a graphical user interface.

To enable this, head to app settings (Cmd / Ctrl + ,) and check the box to run the LLM server on login.

To start the service use: `Lms server start`

Your latest server state will be restored.

Open WebUI



Hello, CESARIUX2596

Give me ideas
for what to do with my kids' art

Overcome procrastination
give me tips

Show me a code snippet
of a website's sticky header

Install Open WebUI

1. Open terminal (Linux) or WSL (Windows)

   Apple silicon still in beta, must be build from scratch, follow installation guide:

   https://docs.vllm.ai/en/latest/getting_started/installation/cpu.html

2. Create a new environment for the dependencies

   ```
   conda  create -n open-webui python=3.12 -y
   ```

3. Install the package

   ```
   pip install open-webui
   ```

4. Serve the service

   ```
   open-webui serve
   ```

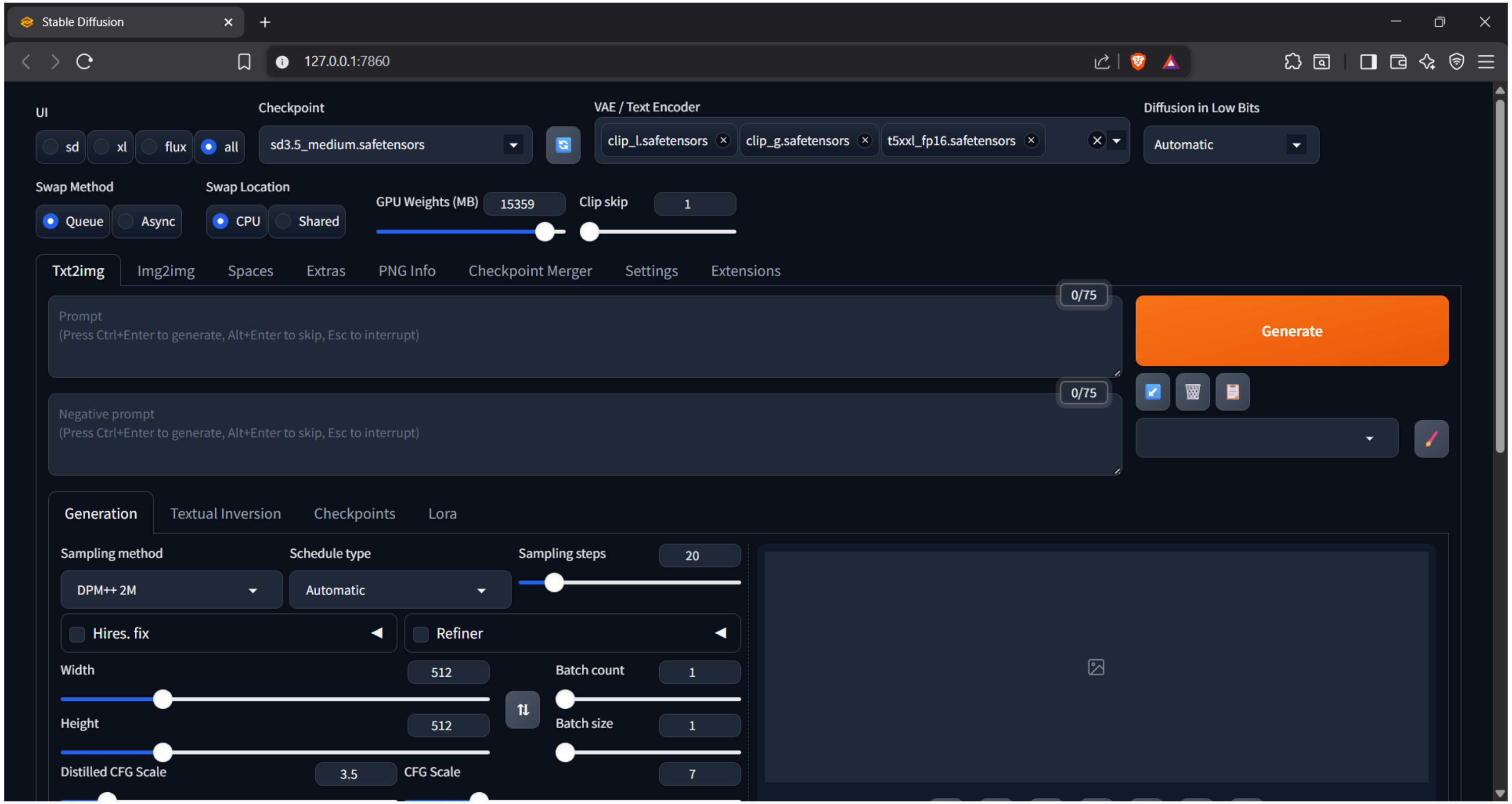
5. Open the instance

   http://localhost:8080/

Why Open WebUI?

Aspect	Pros ✅	Cons ⚠️
Ease of Use	Modern, intuitive web interface with chat history, markdown, images, and tools	Requires external backend (Ollama, vLLM, LM Studio, or OpenAI) to function
Model Library	Supports any model exposed by connected backends (Ollama, HF, Groq, LM Studio, etc.)	Cannot host or serve models itself — UI only
API Compatibility	Fully open, connects to almost everything.	Feature parity depends on the connected backend’s API implementation
Performance	Lightweight frontend; supports streaming responses and multi-backend routing.	Performance limited by backend throughput (no direct inference optimization)
Privacy	100 % self-hosted; can run entirely offline with local backends	Multi-user auth and roles require manual setup or external proxy
Deployment	Docker-ready, cross-platform, integrates easily with vLLM/Ollama/LM Studio	Needs manual network configuration (ports, SSL, reverse proxy for WAN access)

Stable Diffusion Forge



UI: ○ sd ○ xl ○ flux ● all

Checkpoint: sd3.5_medium.safetensors

VAE / Text Encoder: clip_l.safetensors × clip_g.safetensors × t5xxl_fp16.safetensors ×

Diffusion in Low Bits: Automatic

Swap Method: ● Queue ○ Async

Swap Location: ● CPU ○ Shared

GPU Weights (MB): 15359

Clip skip: 1

Prompt
(Press Ctrl+Enter to generate, Alt+Enter to skip, Esc to interrupt)
0/75

Negative prompt
(Press Ctrl+Enter to generate, Alt+Enter to skip, Esc to interrupt)
0/75

Sampling method: DPM++ 2M

Schedule type: Automatic

Sampling steps: 20

□ Hires. fix

□ Refiner

Width: 512

Height: 512

Batch count: 1

Batch size: 1

Distilled CFG Scale: 3.5

CFG Scale: 7

Install Open WebUI

1. Open terminal (Linux) or WSL (Windows) and clone the following repo:

   https://github.com/lllyasviel/stable-diffusion-webui-forge

2. Create a new environment for the dependencies

   ```
   conda  create –n df-forge python=3.10 –y
   ```

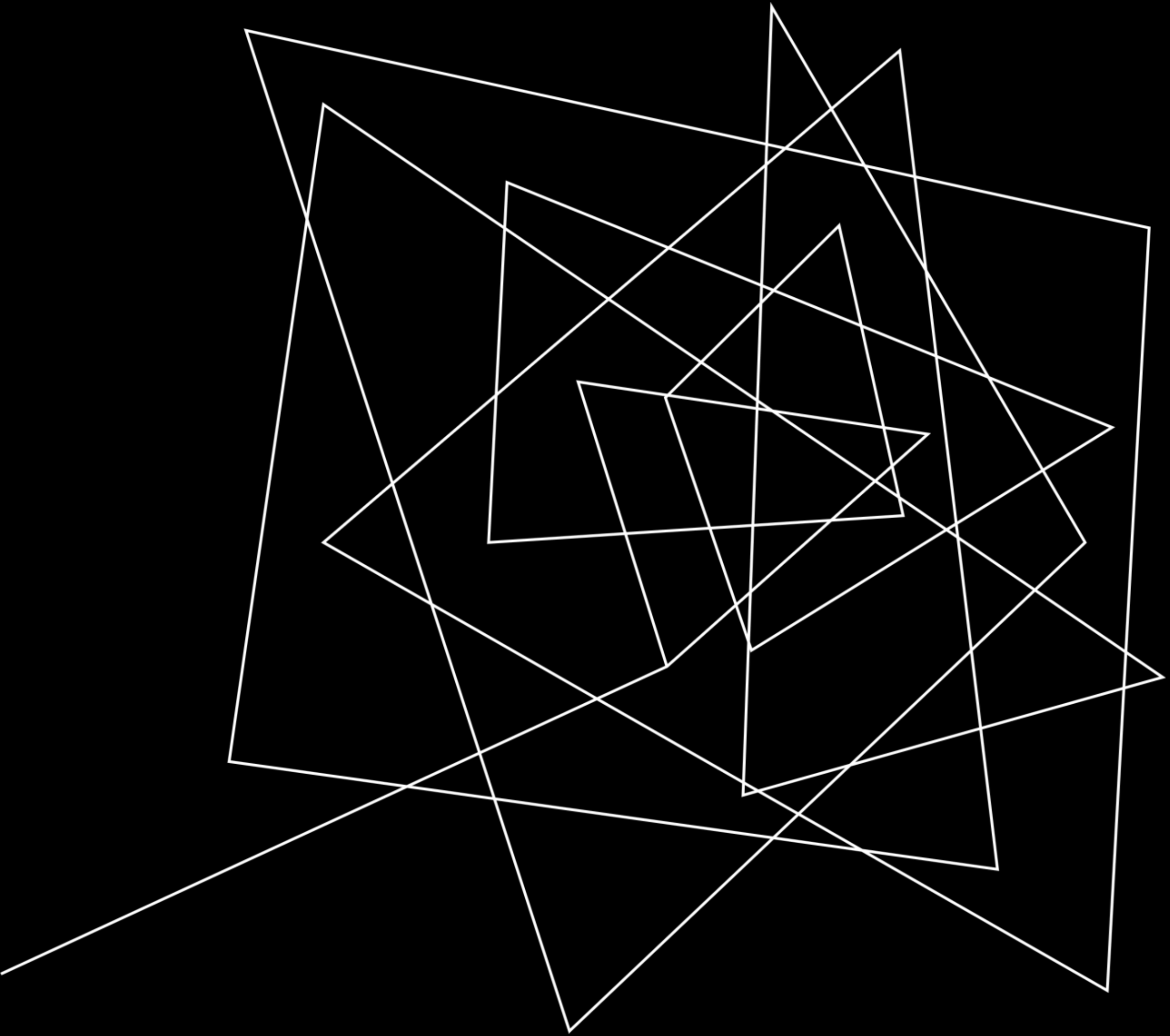
3. Enter the repo

   ```
   cd stable-diffusion-webui-forge
   ```

4. Execute webui-user.bat for windows and webui-user.sh for linux, MacOS must follow instructions: https://github.com/AUTOMATIC1111/stable-diffusion-webui/wiki/Installation-on-Apple-Silicon

Why Open SD-Forge?

Aspect	Pros ✅	Cons ⚠️
Ease of Use	Familiar AUTOMATIC1111-style interface with tooltips, live previews, and integrated extensions.	Some settings (optimization flags, GPU modes) require manual tuning for best performance.
Performance	Noticeably faster generation and lower VRAM use — optimized for mid-range GPUs (6–12 GB).	Limited performance gain on very high-end GPUs; experimental features may affect stability.
Model & Extension Support	Fully compatible with ControlNet, IP-Adapter, Tiled VAE, and most AUTOMATIC1111 extensions.	Some legacy or unmaintained extensions may fail due to codebase differences.
Memory Efficiency	Excellent VRAM optimization — supports higher resolutions and larger batches before OOM.	On laptops or non-NVIDIA GPUs, memory optimization requires manual configuration.
Customization	Supports advanced scripting, prompt editing, and workflow automation (Forge Scripts).	Complex UI — newcomers may find the depth overwhelming at first.
Deployment	Works cross-platform, easy to run via Docker or local Python; active GitHub community with frequent updates.	Rapid development can cause temporary instability between versions or require reinstallation.



Best Practices and Trends

Best Practices and Trends

🧩 **Infrastructure & Deployment**

- Use Docker (and Compose) for reproducible, isolated environments.
- Add reverse proxies (NGINX) for secure routing & HTTPS.
- Expose APIs via FastAPI or Express for lightweight service layers.

⚙️ **Optimization & Maintenance**

- Prefer quantized / LoRA models for efficient VRAM use.
- Log throughput (tokens/s), latency, and power draw to track efficiency.
- Version control configs and adapters; use local cache mirrors for models.

🔒 **Security & Access**

- Keep inference endpoints private — protect with auth or VPN.
- Store data encrypted; disable telemetry in local deployments.

Best Practices and Trends

- **Look Online — Stay Ahead of the Curve**

Social & Community Spaces

- **X (Twitter):** follow researchers, open-source devs, and orgs like *vLLM, Hugging Face, MistralAI, Qwen Team, StabilityAI*.
- **Reddit:** subs like r/LocalLLaMA, r/LocalAI, r/MachineLearning, and r/StableDiffusion — real-world setups, benchmarks, and leaks appear here *first*.
- **Discord:** join project servers (vLLM, LM Studio, Ollama, ComfyUI) — fast support, beta builds, and plugin ecosystems evolve there.

Technical Knowledge Hubs

- **GitHub** — watch repos, track commits, open issues to see what's changing.
- **ArXiv & Papers with Code** — discover *emerging architectures* and *quantization tricks*.
- **YouTube / Medium / Substack** — tutorials, config guides, and experimental reports.

💭 **Trendspotting Tips**

- Follow *contributors*, not just companies — innovation often comes from individuals.
- Save configs, links, and papers in a personal knowledge base (Obsidian, Notion, etc.).

Best Practices and Trends

Cesar Torres

PhD Candidate, Division of Graduate Studies.

TECNM / Tijuana Institute of Technology.

- 📧 Email: cesartorres@tectijuana.mx.
- 🌐 Website: https://cesartorres.org.
- 🐙 GitHub: https://github.com/CESARIUX2596.