Reference Architecture: A Sovereign Chatbot on RLC Pro AI

1. What you will build, and why it's sovereign

A **sovereign chatbot** is a conversational AI stack you own end to end: the model weights, the runtime serving them, the interface in front of them, and the data flowing through it. Nothing depends on a third-party API that can be rate-limited, repriced, or cut off.

This architecture delivers five sovereignty properties, each one the result of a specific design choice:

Property	What it means	This design
Own the model	Open weights you hold, not a vendor API	Model baked into the container image; runs locally
Own the runtime	You control where and how inference runs	RLC Pro AI OCI container — runs on any OCI host
Own the interface	The chat UI is software you run, not someone's SaaS	Stock open-source Open WebUI, self-hosted beside the model
Keep the data	No conversation reaches a third party	Chats stored in your own deployment; no external service in the path
No lock-in	Portable across hardware and clouds	Two OCI containers → bare metal *or* any neo cloud

Why RLC Pro AI containers specifically. RLC Pro AI is CIQ's hardened, security-maintained Rocky Linux 9 AI base, shipped as an OCI image. Three benefits make it the right foundation:

- **Pre-validated and hardened** — a maintained base OS with the AI stack already integrated, not a pile of dependencies you assemble and audit yourself.
- **Grab-and-go reproducibility** — one image, versioned and pinned. With the inference engine baked in, "deploy" is a pull, not a build-from-scratch.

- **Run anywhere OCI runs** — the same image deploys to bare metal, your own Kubernetes, or a serverless neo cloud (a GPU-first serverless provider), so no single provider can hold your stack hostage.

This document builds the chatbot on **Modal** (a serverless GPU cloud) for a concrete, reproducible target. Modal is the one swappable layer — anywhere that runs OCI containers on GPUs works the same way.

2. Architecture at a glance

The system is **two tiers**: a chat frontend on cheap CPU in front of a GPU backend that scales to zero. Each later section covers one box of this diagram.

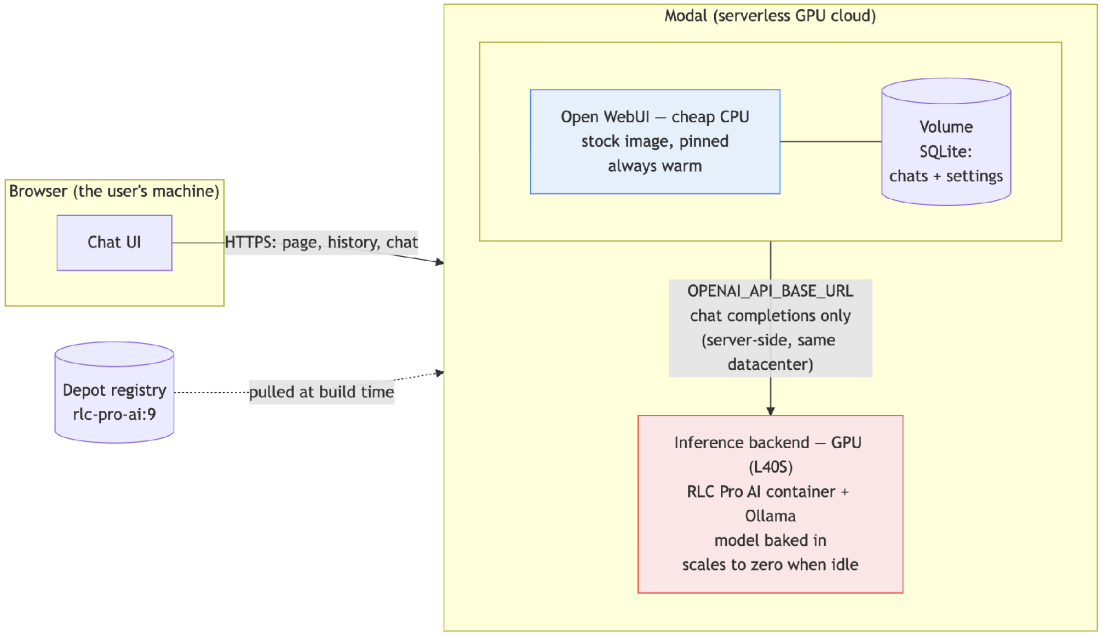

Figure 1. The two-tier architecture. The browser talks only to the self-hosted Open WebUI frontend, which sends real chats — and nothing else — to the scale-to-zero GPU backend running the RLC Pro AI container.

Open WebUI talks to the **GPU backend** over the OpenAI-compatible API, and only a real chat completion crosses that hop. The GPU is never reachable from the browser, and it stays scaled to zero until someone sends a message.

Why split the tiers this way? Three reasons:

- **A visitor who never chats never costs GPU time.** Loading the page, reading past conversations, and populating the model dropdown are all served on CPU. You don't pay ~$2/hr to serve a favicon. (How: §4b.)
- **Cold starts are handled server-side.** The GPU cold start (measured at 1–3 minutes) happens behind Open WebUI's own backend, inside one long-lived

HTTP request. The browser never races a cold cross-origin request that times out.

- **The backend is a standard OpenAI-compatible endpoint.** LM Studio, `curl`, or your own app can point at the same URL the frontend uses — the frontend is one client among many.

3. The container image

The choices that make the backend sovereign all happen at **image build time**. The image is a stack of layers on top of the RLC Pro AI base:

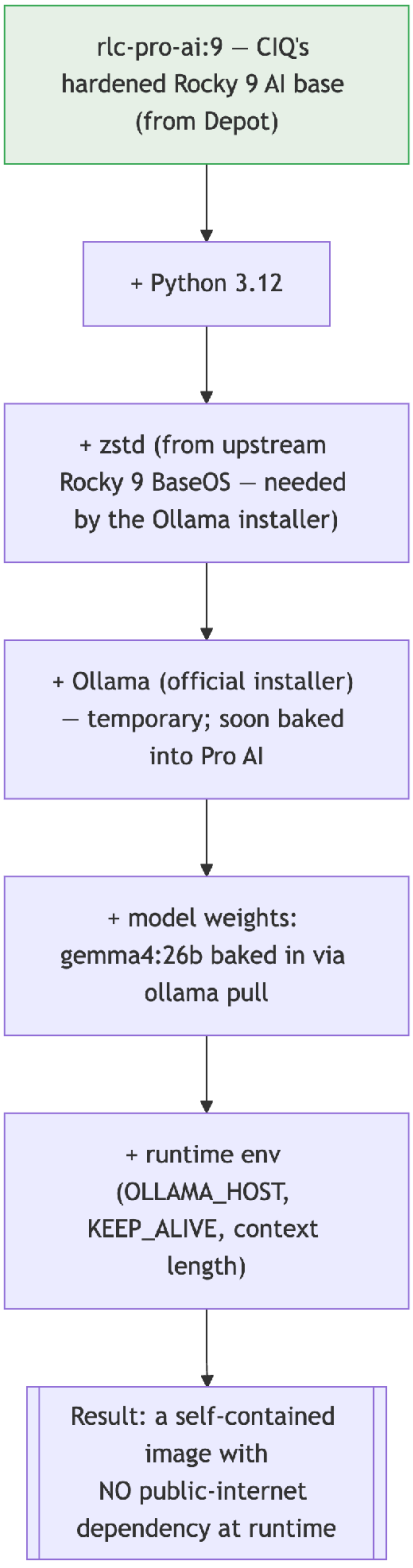
rlc-pro-ai:9 — CIQ's hardened Rocky 9 AI base (from Depot)
+ Python 3.12
+ zstd (from upstream Rocky 9 BaseOS — needed by the Ollama installer)
+ Ollama (official installer) — temporary; soon baked into Pro AI
+ model weights: gemma4:26b baked in via ollama pull
+ runtime env (OLLAMA_HOST, KEEP_ALIVE, context length)
Result: a self-contained image with NO public-internet dependency at runtime

Figure 2. The container image. Each layer adds one capability on top of the RLC Pro AI base; the model weights are baked in, so the running container has no public-internet dependency.

The model weights are **baked into the image**, not pulled at startup. That is what lets the running container be fully offline — air-gap-friendly, reproducible, and with no runtime dependency on a model host that could disappear. It is the difference between "sovereign" and "a thin client for someone else's download server."

The full image definition, with copy-paste code, is in the companion tutorial (*Deploy an RLC Pro AI Inference Backend on Modal with Ollama*).

4. The two tiers in detail

4a. Inference backend (GPU tier)

The GPU container runs the RLC Pro AI image, starts Ollama, and serves chat completions.

- **What it serves:** `POST /v1/chat/completions` (OpenAI-compatible, streaming); a static `/v1/models` for external OpenAI clients that list models on connect; and a `/healthz` endpoint that returns the running container's OS identity (proving it really is the Pro AI base — see §7).
- **Two server-enforced invariants:** Whatever a client sends, the backend pins the `model` field to the baked-in model, and injects the system prompt when the request doesn't carry one. Branding and behavior can't be overridden from the client side, and no request can reference a model that isn't in the image.
- **Why a GPU, and which one:** `gemma4:26b` is a 25B sparse-MoE model. It needs the VRAM to hold all experts resident and the memory bandwidth to decode fast. The **L40S** (48 GB, 864 GB/s) does both with headroom. The smaller L4 (24 GB, 300 GB/s) can't actually hold it at runtime — the model spills to CPU and crawls.
- **Why it scales to zero:** GPU time is the expensive resource. With no minimum, an idle backend costs nothing; it cold-starts on the first chat and stays warm for a configured window (20 min here) so a session of back-and-forth doesn't re-boot between messages.
- **Readiness:** Modal routes no traffic to a container until its startup hook finishes. The backend uses that hook to load the model into VRAM *before* declaring ready — so "a worker is serving" means "the model is resident," not "a container booted but still has to load weights on the first request."
- **Fully offline at runtime.** Weights, engine, and OS all come from the image (§3); the running container makes no outbound calls.

4b. Chat frontend (CPU tier)

The frontend is **stock Open WebUI**, the most widely deployed open-source chat interface, run from its official image at a pinned version with `OPENAI_API_BASE_URL` pointed at the GPU backend. It is configuration, not code.

- **What it provides out of the box:** conversation sidebar and history, Markdown and code rendering, reasoning/thinking display, mobile-responsive layout, prompt presets, and per-user settings — ChatGPT-class UX with no bespoke frontend to maintain.
- **The one piece of server-side state:** Chats and settings live in SQLite on a persistent volume. That state is inside your deployment: no external database, no analytics service, no third party in the path. SQLite tolerates exactly one writer, so this tier is pinned to a single container.
- **Configured for the sovereign posture:** telemetry disabled (three separate phone-home paths switched off), offline mode on, plugins and code execution disabled on the public demo URL, and no local embedding model loaded at startup.
- **The model dropdown works while the GPU is cold.** Open WebUI is told the model list up front instead of discovering it, so the UI is fully populated with **zero** upstream calls — nothing but an actual chat message ever touches the GPU.
- **Licensing.** Open WebUI ships under BSD-3 plus a branding-retention clause: free to self-host, modify, and use commercially, but deployments above 50 users may not remove its branding without an enterprise license. This design retains the stock branding, staying clear of the clause. If white-labeling matters later, LibreChat (MIT) is the drop-in alternative — it speaks the same OpenAI API, so only this tier changes.

Why adopt a frontend instead of writing one. A hand-rolled UI is a permanent maintenance obligation with no sovereignty benefit: a self-hosted open-source frontend keeps the same no-vendor-in-the-path property and adds the features users expect.

5. Request flow: what happens on a chat

This sequence shows the full path of one message, including a cold start.

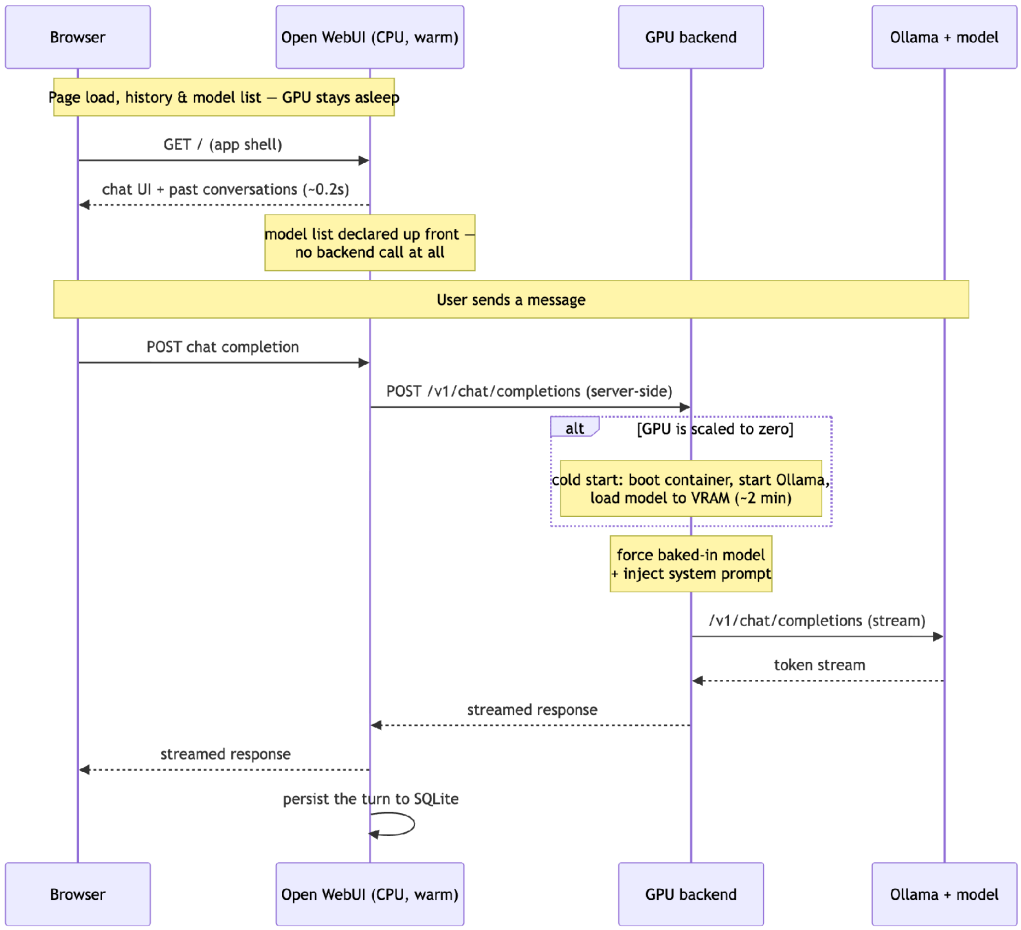

Figure 3. A chat request end to end. Page loads and model-list lookups never wake the GPU; only the chat call crosses to the GPU, with the cold start absorbed server-side.

The top exchange (page load, history, model list) never touches the GPU — that's what keeps a casual visitor free. Warm, a page load lands in about 0.2 seconds. The chat exchange streams from the frontend to the GPU; on a cold backend, the first token arrives after container boot and weight loading (measured at roughly 2.5 minutes end to end), and subsequent messages in the session answer in seconds. Every hop streams — nothing buffers the full response before passing it on.

6. Build it yourself, progressively

You can build this in two stages, each independently runnable. Stop at Stage 1 if an API is all you need.

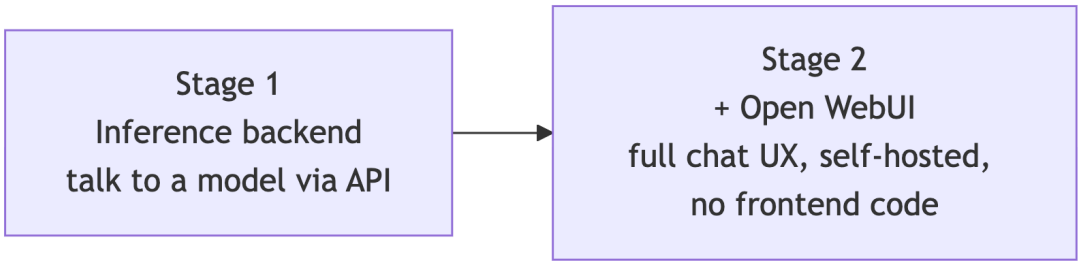

Figure 4. The two build stages. Stage 1 stands alone as an API; Stage 2 adds the chat frontend on top of it.

Stage 1 — Inference backend

Deploy the RLC Pro AI image with Ollama and expose its API — the companion tutorial covers this end to end. (The tutorial's minimal backend loads the model on the first request and uses a 5-minute warm window; this architecture's reference implementation adds the readiness preload from §4a and a 20-minute window on top.) You can verify Stage 1 with a single `curl` to `/healthz` (§7) before building anything on top of it.

Capability gained: a self-hosted, OpenAI-compatible endpoint answering on a GPU you control.

Stage 2 — Add the chat frontend

Run the official Open WebUI image as a second Modal container, mount a volume for its SQLite state, and point it at the Stage 1 backend. No application code — the whole tier is an image plus environment:

```
@app.cls(

image=modal.Image.from_registry(f"ghcr.io/open-webui/open-webui:{VERSION}"),
    volumes={"/data": modal.Volume.from_name("openwebui-data",
                                             create_if_missing=True)},
    scaledown_window=3600,       # cheap CPU; stays warm
    max_containers=1,            # SQLite tolerates exactly one writer
)
class OpenWebUI:
    @modal.enter()
    def start(self):
        # OllamaBackend = the Stage 1 class; Modal resolves its URL at
runtime.
        os.environ["OPENAI_API_BASE_URL"] = OllamaBackend().web.get_web_url()
+ "/v1"
        subprocess.Popen(["bash", "start.sh"], cwd="/app/backend")

    @modal.web_server(port=8080, startup_timeout=300)
    def ui(self):
        pass
```

Set the sovereign-posture environment (§4b) on the image: `DATA_DIR` pointing at the volume mount, telemetry off, offline mode on, plugins and code execution off, and the model list declared up front (`OPENAI_API_CONFIGS`) so the dropdown never calls upstream.

Capability gained: a complete, self-hosted chat product — history, Markdown, mobile — with no vendor in the path.

7. Verifying the sovereign base (provenance)

A sovereign stack should be able to *prove* what it's running. The backend's `/healthz` reads the live container's OS identity and returns it:

```
curl https://YOUR-ENDPOINT.modal.run/healthz
```

```
{
  "ok": true,
  "model": "gemma4:26b",
  "image": {
    "pretty_name": "Rocky Linux from CIQ Pro AI 9.7",
    "variant": "Pro AI",
    "dnf_repos": ["ciq-depot-client.repo", "..."]
  }
}
```

`pretty_name` and `variant` come straight from `/etc/os-release` *inside the running container* — confirming it is the official RLC Pro AI base, not a stock Rocky image with a model bolted on.

8. Design decisions and trade-offs

Decision	Choice	Trade-off
Chat frontend	Stock Open WebUI, pinned	A dependency and a branding-retention license clause; in exchange, ChatGPT-class UX and no frontend to maintain
Conversation state	SQLite on a persistent volume	One stateful tier, capped at one writer; in exchange, real chat history — and the data still never leaves your deployment
API surface	Chat served straight from the GPU	An external client polling `/v1/models`

Decision	Choice	Trade-off
	backend; no gateway tier	wakes the GPU; in exchange, one fewer service to run and explain
GPU scaling	Scale to zero	First chat pays a cold start; in exchange, an idle demo is free
GPU choice	L40S (oversized for the weights)	Pay ~$2/hr warm for headroom; in exchange, responses feel fast (bandwidth, not capacity, is the bottleneck)
Model delivery	Baked into the image	Larger image, rebuild to change models; in exchange, fully offline runtime + reproducibility
Inference engine	Ollama	A temporary install layer; in exchange, zero serving code — and the layer disappears when Pro AI bakes Ollama in

9. Production hardening (beyond the demo)

This reference architecture deploys a public demo. For a real deployment, layer on:

- **Auth.** Open WebUI ships multi-user accounts, roles, and SSO/OIDC — the demo runs with auth off for frictionless access. Turn it on before exposing the URL. (Enable it from the start: switching a running deployment from auth-less to authenticated requires a fresh database.) Protect the GPU endpoint separately with `requires_proxy_auth=True`.
- **Rate limiting.** Add it in front of the chat endpoint — the public URL is otherwise an open inference endpoint.
- **Scaling.** Set `min_containers` to keep a warm GPU if cold starts are unacceptable; cap `max_containers` for cost control.
- **Backups.** The volume now holds conversation history; treat it as data, not cache.
- **Image pinning.** Pin the Pro AI base *and* the frontend by digest for reproducibility. Frontend upgrades run irreversible database migrations, so treat version bumps as a tested change, not a floating tag.

- **Observability.** Log request metadata (never conversation content) for capacity planning.

10. Running on your own Kubernetes cluster

Modal is the one swappable layer (§1). Both OCI images — the RLC Pro AI base with Ollama and the model weights baked in, and stock Open WebUI — run unchanged on any Kubernetes cluster with GPU nodes. What does not port is the Modal deploy script: it is Modal-specific, and Kubernetes manifests take its place.

What you write instead:

- **Backend Deployment.** A GPU resource request (`nvidia.com/gpu: 1`) on a cluster running the NVIDIA device plugin on its GPU nodes.
- **Backend Service.** A ClusterIP on port `11434`, so the GPU is reachable from the frontend and from nothing else.
- **Frontend Deployment.** Open WebUI with a PersistentVolumeClaim for its SQLite data, pinned to one replica — SQLite tolerates exactly one writer (§4b).
- **Ingress or LoadBalancer.** For the frontend only, with auth in front of it (§9).
- **imagePullSecret.** Credentials for the Depot registry that serves the Pro AI image.

```
# sketch — backend Deployment, GPU request
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: ollama
          image: <your-registry>/rlc-pro-ai-ollama:<tag>
          ports:
            - containerPort: 11434
          resources:
            limits:
              nvidia.com/gpu: 1
```

Scale-to-zero does not come along for the ride. Kubernetes will not scale a Deployment to zero on its own — its floor is one replica. Either keep one backend replica warm and pay for the GPU continuously, or run an autoscaler that can reach zero, such as KEDA scaling on request metrics or a queue, and accept the same cold start §5 measures.

What stays the same: every sovereignty property in §1 — own the model, own the runtime, own the interface, keep the data, no lock-in. The Kubernetes path strengthens the last one: it takes the neo cloud out of the request path entirely, leaving nothing between your users and hardware you control.

Treat this section as a sketch. It has not been run end to end — the manifests above are a starting point to adapt to your cluster's storage classes, ingress controller, and GPU node labels, not a tested deployment.

11. Summary

A sovereign chatbot is three moves:

- **Own the model + runtime** — bake an open model into the RLC Pro AI container.
- **Make idle free** — serve it from a GPU that scales to zero.
- **Own the interface + keep the data** — put a self-hosted open-source frontend in front of it.

Because it's all OCI containers, the same stack runs on bare metal or any neo cloud — which is the whole point: no lock-in, anywhere you want to run it.

Start at the companion tutorial (Stage 1), then add Stage 2 from §6.