Deploy an RLC Pro AI Inference Backend on Modal with Ollama

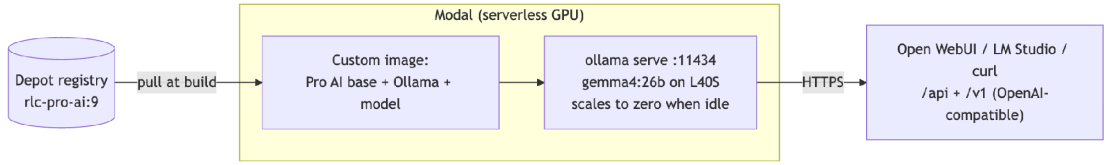
Overview

RLC Pro AI ships as an OCI container image, so you can run CIQ's hardened, pre-validated AI base anywhere that runs containers — including neo clouds. This guide deploys the RLC Pro AI image to **Modal** (a serverless GPU cloud) and exposes **Ollama** as a model-inference endpoint. The result is a self-hosted, OpenAI-compatible API serving an open-weights model (Gemma 4) on a GPU you control.

This is the foundation for a **sovereign AI stack**: your model, your runtime, no dependency on a third-party model provider. It is also Stage 1 of the companion reference architecture, which turns this working API into a deployed chat product.

What you will build

A public HTTPS endpoint that answers `/api/generate` and `/v1/chat/completions` requests, on a GPU that scales to zero when idle — plus a chat client pointed at it.



Deploy the RLC Pro AI image to Modal, layer on Ollama and a model, and expose it as an OpenAI-compatible endpoint.

Prerequisites

- **A CIQ portal account with RLC Pro AI entitlement.** Confirm at https://portal.ciq.com → **My Products** → **RLC Pro AI**. This grants pull access to the Depot container registry.
- **A Modal account** (https://modal.com) with a payment method on file. GPU functions require a card even when you are spending credits.
- **Python 3.10+** locally, with the Modal CLI:

  ```
  pip install modal
  modal setup        # opens a browser to authenticate this machine
  ```

- **Depot pull credentials** for the RLC Pro AI registry (username + token). Find these in the CIQ portal under **Deploy → Containers** for RLC Pro AI.
- **Docker** (optional) — used to run the Open WebUI chat client in step 6.

Procedure

1. Store your Depot credentials as a Modal secret (one-time setup)

Modal pulls private images using a named secret. Create one called `ciq-depot` holding your registry username and token. Modal's registry auth expects the keys `REGISTRY_USERNAME` and `REGISTRY_PASSWORD`:

```
modal secret create ciq-depot \
  REGISTRY_USERNAME='your-depot-username' \
  REGISTRY_PASSWORD='your-depot-token'
```

The secret is encrypted at rest and injected only at build time — it never lands in your source.

2. Define the container image

Create `inference_backend.py` with the following code. The image starts from the official RLC Pro AI base and layers on Ollama plus the model weights:

```
import subprocess
import modal

MODEL = "gemma4:26b"

app = modal.App("rlc-proai-ollama")

image = (
    modal.Image.from_registry(
        # The official RLC Pro AI image — CIQ's Rocky Linux 9 + AI base —
        # pulled from the private Depot registry using your Modal secret.
        "depot.ciq.com/rlc-ai-9/rlc-9-ai-oci-images/rlc-pro-ai:9",
        add_python="3.12",
        secret=modal.Secret.from_name("ciq-depot"),
    )
    .run_commands(
        # The Ollama installer needs `zstd`, which the RLC Pro AI image's
depot
        # repo doesn't carry until `depot enable`. Pull it from upstream
        # Rocky 9 BaseOS at build time (RLC is Rocky-9-compatible). Runtime
        # stays fully offline.
        "dnf install -y --repofrompath "

"'rocky-baseos,https://dl.rockylinux.org/pub/rocky/9/BaseOS/x86_64/os/' "
        "--nogpgcheck zstd && dnf clean all",
        # Install Ollama with the official installer.
        "curl -fsSL https://ollama.com/install.sh | sh
",        # Bake the model into the image so the container has no
public-internet
        # dependency at startup. `ollama serve` runs just long enough to
```

```
pull.
        f"ollama serve > /tmp/ollama-build.log 2>&1 & "
        f"sleep 10 && ollama pull {MODEL}",
    )
    .env({
        "OLLAMA_HOST": "0.0.0.0:11434",   # listen on all interfaces
        "OLLAMA_KEEP_ALIVE": "-1",        # keep the model resident while
warm
    })
)
```

3. Expose Ollama as a web endpoint

Append the following class to `inference_backend.py`. `@modal.web_server` turns the process listening on port 11434 into a public HTTPS endpoint, exposing Ollama's full API, including the OpenAI-compatible `/v1` routes:

```
@app.cls(
    image=image,
    gpu="L40S",            # 48 GB VRAM / 864 GB/s — fits gemma4:26b with
headroom
    scaledown_window=300,  # stay warm 5 min after the last request, then
scale to zero
)
@modal.concurrent(max_inputs=10)
class OllamaBackend:
    @modal.web_server(port=11434, startup_timeout=180)
    def serve(self):
        # Non-blocking: Modal waits for port 11434 to accept connections,
        # then routes traffic to it. The baked-in model loads into VRAM on
        # the first request.
        subprocess.Popen(["ollama", "serve"])
```

That is the entire backend — about 40 lines including comments.

4. Deploy

```
modal deploy inference_backend.py
```

Modal builds the image (first build pulls and bakes the model — a few minutes), then prints the public URL of your endpoint, e.g.:

```
https://your-workspace--rlc-proai-ollama-ollamabackend-serve.modal.run
```

Copy this URL — in every command below, replace `https://YOUR-ENDPOINT.modal.run` with the exact URL Modal printed.

5. Verify the endpoint

Call the endpoint. The **first** request cold-starts the GPU and loads the model (measured at 1–3 minutes) — the command sits with no output until then, which is normal; warm requests are fast.

Ollama native API:

```
curl https://YOUR-ENDPOINT.modal.run/api/generate -d '{
  "model": "gemma4:26b",
  "prompt": "In one sentence, what is sovereign AI?",
  "stream": false
}'
```

OpenAI-compatible API:

```
curl https://YOUR-ENDPOINT.modal.run/v1/chat/completions \
  -H "Content-Type: application/json" -d '{
  "model": "gemma4:26b",
  "messages": [{"role": "user", "content": "Hello from RLC Pro AI"}]
}'
```

A JSON completion confirms the endpoint is live.

6. Connect a chat frontend

Your endpoint speaks the OpenAI-compatible API, so any compatible chat app can talk to it — no vendor account, no third-party round-trip. Point the app's base URL (sometimes labeled "API host" or "endpoint") at your endpoint's `/v1` route, and use:

- **Base URL:** `https://YOUR-ENDPOINT.modal.run/v1`
- **Model:** `gemma4:26b`
- **API key:** any non-empty string — Ollama ignores it unless you enabled proxy auth

Open WebUI is the recommended pairing: self-hosted, container-native, and the most widely deployed open-source chat interface. To try it locally against the endpoint you just deployed:

```
docker run -d -p 3000:8080 \
  -e ENABLE_OPENAI_API=true \
  -e OPENAI_API_BASE_URL='https://YOUR-ENDPOINT.modal.run/v1' \
  -e OPENAI_API_KEY='any-non-empty-string' \
  -v open-webui:/app/backend/data \
  ghcr.io/open-webui/open-webui:v0.10.2
```

Open http://localhost:3000, pick `gemma4:26b`, and chat. Two alternatives if it doesn't fit: **LM Studio**, a cross-platform desktop app, is the fastest path to a single-user demo; **LibreChat** is the MIT-licensed choice when multi-user and white-labeling matter.

Caveat — idle clients can wake a scaled-to-zero GPU. OpenAI-compatible clients poll `/v1/models` on connect, and that poll wakes an idle GPU. The companion reference architecture avoids it by declaring Open WebUI's model list up front (`OPENAI_API_CONFIGS`), so nothing but a real chat ever touches the GPU.

Cost and scaling

- **Scale-to-zero** (the default above) bills GPU time per second — an idle backend costs nothing.
- The **L40S** runs ~$2/hr while warm. The cheaper L4 (24 GB) can't hold the 26B model at runtime — it spills to CPU and crawls. The L40S has both the VRAM and the memory bandwidth.
- To avoid the first-request delay from step 5, set `min_containers=1` to keep one GPU warm.

Security

The endpoint is public — anyone with the URL can call it. Before production, require auth: set `requires_proxy_auth=True` on `@modal.web_server` and call with a Modal proxy-auth token, or front the endpoint with your own gateway.

Next steps

- **Build the full chatbot:** see the companion document, *Reference Architecture: A Sovereign Chatbot on RLC Pro AI*.
- **Swap the model:** change `MODEL` and, if larger, the `gpu=` value — the same pattern scales up to frontier-grade open-weights models like Kimi K3 (multi-GPU; the architecture is unchanged).
- **Harden for production:** apply the Security section above, then add rate limiting and a custom domain.