Local AI Deployment and Performance

Where to physically run the LLM endpoint relative to TServer — the tradeoffs, the four topologies, and the sizing recommendations. The REST-over-OpenAI-compatible-POST design makes every topology a one-field configuration change; this page covers when to use which.

AI Integration Local AI Deployment and Performance

Version 10.1.5+

TL;DR

LLM inference is CPU-heavy. A single call on a 7B-class model uses every available core for 1–30 seconds. If the LLM runs on the same host as TServer, the runtime competes with the model for CPU and operator-facing latency suffers. **Run the LLM on a separate GPU machine — or a remote / cloud endpoint — not on the FrameworX/TServer host. Installing models on the FrameworX host is against company recommendation in every scenario: production, demos, evaluation, and testing alike.** A small CPU-only model on the TServer box produces only ~2-4 tokens/sec, too slow even for a demonstration. The REST surface is OpenAI-compatible HTTP — only the `URL` field in `SolutionCapabilities[LocalAI].Settings` changes. For remote and cloud placement specifically, see Remote and Cloud LLM Models.

Why this matters

FrameworX Local AI calls an OpenAI-compatible chat-completions endpoint over HTTP. Both consumption paths — the `ChatRequest` Display action and the `AI.Execute` script API — issue the POST from inside the TServer process or one of its server-domain children (ScriptTaskServer.exe, ReportServer.exe). See Local AI Architecture Reference § *Where Local AI runs* for the per-process breakdown.

What that page does *not* address is the endpoint on the other side of the POST — where the actual LLM model loads, decodes, and generates tokens. That host has different resource profile from TServer:

- **LLM inference saturates CPU.** A modern 7B-class model running CPU-only generates only ~2–4 tokens per second on a typical x64 server — too slow even for a demonstration — and uses every available core to do it. A 50-token response is ~15–25 seconds of 100% CPU; a 200-token narrative is over a minute. There is no "throttle" knob — the model uses what it has.
- **TServer also wants CPU.** The runtime polls devices, runs scripts, evaluates expressions, dispatches alarms, and serves clients. None of those are CPU-heavy individually, but they are latency-sensitive: a missed scan slows control loops, a delayed heartbeat shows as *Disconnec ted* on operator panels.
- **Co-location starves both.** When TServer and the LLM share a host, every active LLM call effectively pauses the runtime for the duration of inference. Symptoms: Property Watch panels freeze, the DesignerTServer heartbeat times out at 5 seconds, the HTML5 client feels sluggish, alarm reactions delay by seconds.

This is not a FrameworX limitation — it is the nature of CPU-bound large-model inference. The fix is topological: move the inference workload off the runtime's host entirely — onto a separate GPU machine or a remote / cloud endpoint.

The four topologies

Topology	Hardware	Latency profile	When to choose
1. Same host — LLM runs on the TServer machine	16+ GB RAM, modern 8+ core CPU. GPU optional but unused if absent.	Inference latency native (no network). TServer contention during every call; CPU-only runs at only ~2-4 tok/s.	**Never — against company recommendation.** Do not install or run a model on the FrameworX/TServer host for any purpose — production, demo, evaluation, or testing. The model competes with the runtime for CPU, and CPU-only inference is too slow (~2-4 tok/s) even for a demonstration. Run the model on a separate GPU machine (topology 3) or a cloud endpoint (topology 4) instead.
2. Separate VM on same hardware — LLM in a sibling VM	32+ GB RAM split, host hypervisor pins CPU cores per VM.	Inference latency native (loopback or same-host network). TServer isolated from LLM CPU use, but the LLM still shares the physical CPU unless a GPU is passed through.	Existing hardware budget allows separation; ops team manages one box. A middle ground when adding a new physical host is friction — but without a passed-through GPU the model is still CPU-bound and slow.
3. Separate physical host — dedicated LLM server	Dedicated box, 16–64 GB RAM, GPU strongly recommended (cuts latency 3–5×). Can be a gaming PC, dev workstation, refurbished workstation, or a small server.	Inference latency native to that host; +1–3 ms network. TServer never sees LLM load. Multiple FX runtimes can share one LLM host.	**Recommended for demos, training, and production.** Regular AI use. Multiple solutions sharing one LLM. GPU acceleration desired.
4. Cloud LLM endpoint — off-premises managed model	Outbound HTTPS only. SecuritySecrets-backed API key.	Network round-trip dominates (50–500 ms) for short prompts; inference itself can be 2–10× faster than CPU-only local. Throughput limited by provider rate-limits.	Strong external LLM justified (GPT-class quality, multi-modal needed). No air-gap constraint. Per-call cost acceptable. Sensitive data redacted before send. See Remote and Cloud LLM Models.

The REST surface enables all four

Local AI calls an OpenAI-compatible `chat/completions` endpoint over HTTP. The TServer process opens the TCP connection; the endpoint can be on a separate GPU machine at `192.168.1.50`, on a dedicated server at `llm-host.lan`, or at `https://api.example-provider.com` — replace 192.168.1.50 with the IP of the machine running your model (the model must not run on the FrameworX/TServer host). From FrameworX's side, all cases are the same code path with a different `URL` string.

Switching topology after the solution is built is a single field edit on the AI Engine tile in **Solution Capabilities**. The JSON shapes for each topology — remote Ollama, OpenAI-compatible cloud with Bearer token, endpoint with extra headers — are documented on Local AI Configuration Reference § *Pointing at a different LLM endpoint*, and the remote / cloud guidance on Remote and Cloud LLM Models.

Authentication and credential handling for non-local endpoints is covered on SecuritySecrets Authentication for Local AI. The short version: API keys go in the SecuritySecrets vault and are referenced from the `Authorization` or `Headers` field via `/secret:<Name>` tokens — they never appear in plain text in the solution.

Sizing recommendations

Separate GPU host (demos, training, production) — recommended

- **Model:** `qwen2.5:7b-instruct` (~4.7 GB) — the FrameworX-recommended default quality / reasoning tier, the floor even for demos — or `qwen2.5:32b-instruct` for maximum reasoning if the LLM host has the GPU and RAM. Tool-call reliability (the `ChatRequest` action's autonomous tool dispatch) is solid at 7B and improves further at 32B.
- **RAM on the LLM host:** 16 GB minimum for 7B, 32 GB+ recommended for the 32B tier. The model loads fully into RAM (or VRAM) and stays warm.
- **CPU on the LLM host:** 8+ cores if no GPU; the inference parallelizes across all of them — but CPU-only is still only ~2-4 tok/s, so a GPU is strongly preferred.
- **GPU on the LLM host:** Strongly recommended. An entry consumer GPU (RTX 3060 / 4060 class) typically runs 7B at native human-conversation speed. A 24 GB GPU comfortably runs the 32B tier.
- **Network:** any reliable link, sub-10 ms latency. The Local AI client uses a 60-second per-call wall-clock timeout (configurable via `TimeoutSeconds` in the Settings JSON), so even occasional network blips do not break the surface.
- **Expect:** 1–3 sec per operator chat reply with a GPU. CPU-only on a dedicated host is still ~2-4 tok/s (much better than co-located because TServer is not waiting on CPU, but still too slow for interactive chat — add a GPU).

Cloud topology

- **Model:** match the provider's recommended instruct model. FrameworX requires only OpenAI-compatible chat-completions; the model selection is opaque to the platform. See Remote and Cloud LLM Models.
- **Authentication:** always SecuritySecrets-backed (see referenced page above). Hard-coded API keys are a deployment anti-pattern.
- **Data sensitivity:** the prompt the LLM sees can include live tag values, alarm context, batch IDs — treat that as exfiltrated data. Compliance review before pointing FX at a third-party endpoint.
- **Cost model:** per-token billing. For a busy alarm-narrative workload (one narrative per critical alarm raise, ~150 input + ~80 output tokens), budget by alarm volume.

Same host — against company recommendation

- **Do not use — for any purpose.** Installing or running a model on the TServer box is against company recommendation in every scenario, including quick wiring checks and internal testing. CPU-only inference produces only ~2-4 tokens/sec and competes with the runtime for CPU. For every case — wiring checks included — point the solution at a model on a separate machine (topology 3) or a cloud endpoint (topology 4); only the `URL` field changes.

Decision matrix — pick a topology by workload

Workload	Same host	Sibling VM	Dedicated host	Cloud
Demo / training class	No — too slow, ~2-4 tok/s	Only with GPU passthrough	**Yes**	Yes
Single operator, occasional chat	No — too slow on CPU	Only with GPU passthrough	**Yes**	Fine
Multiple operators, regular chat	No (queueing + CPU starvation)	Acceptable with GPU passthrough	**Yes**	**Yes**
Alarm-narrative on every critical alarm	No (TServer pause-on-narrate)	Acceptable with GPU passthrough	**Yes**	**Yes**
End-of-shift batch summaries	No	Acceptable	**Yes**	**Yes**
Multiple FX solutions sharing one LLM	No	Yes	**Yes**	Yes (rate-limit-aware)
Air-gapped site, IP-sensitive prompts	No	Yes (GPU passthrough)	**Yes**	No
Per-tick reasoning in fast scan loops	No	No (still CPU-bound)	Only with GPU	Only with cloud rate budget

---

Watch for these symptoms when co-located

If you suspect the LLM is stealing CPU from TServer, watch for these in the same-host topology (a reason to move the model off the host onto a separate GPU machine):

- **Designer Property Watch panels stop updating** for tens of seconds at a time, then catch up in bursts. The DesignerTServer ServiceClient heartbeat is timing out at 5 seconds.
- **Alarm acknowledgment lag.** Operator clicks Acknowledge; the alarm state visually updates several seconds later.
- **HTML5 client feels sluggish.** Page refreshes pause; data bindings re-evaluate in chunks.
- **Scan-period overruns** in the runtime log on Device or Script Tasks set to short periods (under 1 second).
- **Synchronous `AI.Execute` from a Script Task hangs the entire ScriptTaskServer.exe** for the call duration. Other tasks queued behind it wait.

None of these are FrameworX defects — they are the operating-system scheduler responding to a CPU-bound peer process. Moving the LLM endpoint to a separate GPU machine (or a remote / cloud endpoint) eliminates all of them.

What defaults ship

- **Fresh customer install of FrameworX 10.1.5+.** The AI Engine endpoint ships a placeholder — `http://192.168.1.50:11434/v1/chat/completions` with `qwen2.5:7b-instruct`, the recommended default. The install may seed a loopback (`127.0.0.1` / `localhost`) value, but that is a placeholder you MUST change to the IP of the machine running your model (the model must not run on the FrameworX /TServer host). 7B expects a GPU — run it on a separate GPU machine. To set the URL or model, edit **Solution Capabilities AI Engine Edit Configuration**.
- **Shipped demo solutions** (BottlingLine ML Demo, SolarPanels MCP Demo, LocalAI KnowledgeGraph Demo) and **the Local AI Chat Example** ship a placeholder Ollama endpoint and model name — you must point the URL at the machine running your model before they work end-to-end. The recommended model is `qwen2.5:7b-instruct` on a separate GPU machine. Pull the model named in the solution's AI Engine settings with `ollama pull <name>` on that host.
- **Production deployments** override at **Solution Capabilities AI Engine Edit Configuration** — pick `qwen2.5:7b-instruct` (or `qwen2.5:32b-instruct`) and point `URL` at a separate GPU machine, a remote Ollama, or a cloud LLM endpoint. The `Settings` JSON written by the dialog takes precedence over the installed default.

See also

- Remote and Cloud LLM Models — running the model off the FrameworX host on a remote or cloud endpoint.
- Local AI Configuration Reference — JSON examples for each topology's `Settings` value (remote Ollama, cloud, custom headers).
- Local AI - First Install Walkthrough — getting Ollama running on the machine that will serve the model.
- SecuritySecrets Authentication for Local AI — storing API keys safely for cloud / remote endpoints.
- Local AI Architecture Reference § *Where Local AI runs* — which FX-side processes initiate the LLM POST.

In this section...