Local AI

FrameworX Local AI is the platform's built-in, on-local-network LLM integration. Operators chat with a local model from Display panels; server-side scripts call the model atomically for narration, classification, translation, and summary tasks.

Version 10.1.5+

FrameworX recommends `qwen2.5:7b-instruct` (Apache 2.0, ~4.7 GB) as the default Local AI model — the best balance of reasoning and reliable JSON tool-call output, and the model used for new solutions, demos, and templates. For real use run `qwen2.5:7b-instruct` on a separate GPU machine — the floor even for demos. CPU-only produces only ~2-4 tokens/sec, too slow even for a demonstration; 3B is at most a last-resort for single-shot atomic tasks, never interactive chat. Local AI runs on your own network with no cloud dependency — on a separate machine, not the FrameworX host. You install Ollama yourself on the machine that will serve the model (FrameworX ships no local installer); see Local AI - Installing Models (Windows, macOS, Linux) for per-OS setup, and the **First Install Walkthrough** child page for what to expect.

> **Recommended default and limited-hardware fallback**
>
> The recommended default is `qwen2.5:7b-instruct` (~4.7 GB, Apache 2.0) — the strongest balance of multi-step reasoning and reliable JSON tool-call output, and the model used for new solutions, demos, and templates. For real use run `qwen2.5:7b-instruct` on a separate GPU machine — the floor even for demos. CPU-only produces only ~2-4 tokens/sec, too slow even for a demonstration; `qwen2.5:3b-instruct` (~2 GB, Apache 2.0) is at most a last-resort for single-shot atomic tasks (reporting, classification, summary), never interactive chat. For maximum reasoning on a strong GPU, `qwen2.5:32b-instruct` is the performance tier. To pull and select a model, run `ollama pull qwen2.5:7b-instruct`, then in Designer go to **Solution Capabilities AI Engine Edit Configuration** and set the **Name** field.

Recommended model — qwen2.5:7b-instruct

This is the model FrameworX recommends as the default, and the one used for new solutions, demos, and templates. It delivers the best balance of multi-step reasoning and reliable JSON tool-call output — 7B is the floor below which the structured tool-call envelope starts to malform. It expects a machine with a GPU; Ollama auto-detects and uses an NVIDIA CUDA or Apple Metal GPU.

Item	Value
Model name	`qwen2.5:7b-instruct` — the exact string that goes in the Local AI `Name` field and the `ollama pull` command.
License	Apache 2.0. Commercial use permitted, no royalty, no per-seat fee. Suitable for distribution with customer solutions.
Size on disk	~4.7 GB (quantized, stored under `%USERPROFILE%\.ollama\models\`).
Why this model	Best tool-call reliability for the chat + MCP-tool surface and the strongest reasoning in its size class. Handles operator chat, alarm diagnosis, complex tool-call chains, translation, and summary tasks.
Hardware	16 GB RAM recommended; a GPU is expected for usable interactive-chat latency (NVIDIA CUDA / Apple Metal auto-detected). Full per-resource breakdown on the Local AI - First Install Walkthrough.
How to install	Install Ollama on the machine that will serve the model (you provide the runtime — FrameworX ships no local installer), then pull the model: `ollama pull qwen2.5:7b-instruct`. Per-OS setup (Windows, macOS, Linux): Local AI - Installing Models (Windows, macOS, Linux); first-install orientation and what to expect: Local AI - First Install Walkthrough.
How to verify	Three checkpoints, in increasing depth: (1) the script's final `Inference returned in N.Ns: 'pong'` line confirms the endpoint responds; (2) the **Status** indicator on the Local AI tile in **Solution Capabilities** (Designer) probes the endpoint every 30 seconds and reports **Reachable** in green; (3) a ChatRequest from any Display panel returns a reply envelope with `status = "ok"` and a populated `text` field. Any of the three failing surfaces the same root cause (Ollama not started, model not pulled, port held by another process).

Limited-hardware fallback — qwen2.5:3b-instruct

On a machine with no GPU or very limited resources, `qwen2.5:3b-instruct` (Apache 2.0, ~2 GB) is the last-resort fallback. It runs on a modern x64 CPU without a GPU and downloads in minutes, but CPU-only produces only ~2-4 tokens/sec — too slow even for a demonstration, and **not for interactive chat**. Reserve it for single-shot atomic tasks only: alarm annotation, translation, classification, and short summaries, where one call does not depend on sustained multi-turn reasoning. For anything interactive — including demos — run 7B on a separate GPU machine instead.

To use it:

1. Pull the model: `ollama pull qwen2.5:3b-instruct`
2. In Designer, go to **Solution Capabilities AI Engine Edit Configuration**.
3. Set the **Name** field to `qwen2.5:3b-instruct` and save.

8 GB RAM minimum; modern x64 CPU sufficient; no GPU required.

Maximum performance — qwen2.5:32b-instruct

For the strongest reasoning and multi-step tool logic on a machine with a strong GPU (roughly the 20 GB VRAM class), `qwen2.5:32b-instruct` (~20 GB) is the performance tier. Pull it with `ollama pull qwen2.5:32b-instruct` and set the **Name** field accordingly. Best run on a dedicated GPU host — see **Running Ollama on a separate host** below.

Choosing a model

The 7B (recommended default) and 3B (limited-hardware fallback) split above covers most cases, with 32B as the maximum-performance tier on a strong GPU. The notes below help match the model to the workload when that split is not the only axis.

Workload	Recommended	Why
Operator chat panel — short conversational prompts, single-turn or light multi-turn.	`qwen2.5:7b-instruct`	7B holds multi-turn context and response structure reliably — the right default for an interactive operator chat panel. Run it on a separate GPU machine; 3B on CPU is too slow even for a demo.
Structured output / tool calling — `AI.Execute` flows that parse JSON envelopes or chain tool calls.	`qwen2.5:7b-instruct` (strongly recommended)	3B can drift on JSON shape under pressure (missing fields, malformed tool-call arguments). 7B holds the contract reliably.
Long context — large UNS summaries, multi-turn history, sizeable system prompts.	`qwen2.5:7b-instruct`	Both qwen2.5 models accept a 32K-token context window, but the 3B's effective reasoning window is narrower. Bias toward 7B as the context grows.
Hardware budget — no GPU, 8 GB free RAM.	`qwen2.5:3b-instruct` (last-resort fallback)	3B runs on a modern CPU without a GPU — use it for single-shot atomic tasks only, never interactive chat. For any chat — including demos — run 7B on a separate GPU machine; CPU-only produces only ~2-4 tokens/sec.
GPU acceleration available.	`qwen2.5:7b-instruct` default; `qwen2.5:32b-instruct` for maximum reasoning on a strong GPU.	Ollama supports CUDA (NVIDIA) and Metal (Apple); ROCm (AMD) is improving — verify driver compatibility before committing.

Other models. Ollama supports many models beyond the qwen2.5 family. Any OpenAI-compatible chat completion model with tool-call support should work; the FrameworX team tests primarily on qwen2.5. Other models may require `Authorization` or response-format tweaks — verify with a smoke test before committing a production solution to an untested model.

Overview

Local AI is shipped as solution infrastructure. There is one LLM endpoint per solution; every consumer in the solution — operator chat, script call, alarm callback, report generator — reaches the same model through the same configuration. Two consumption patterns:

- **Operator chat — ChatRequest action.** A Display button or any interactive control fires a **ChatRequest** Action; the operator's typed query goes to the model and the reply lands on a tag the Display reads. A built-in per-Display-panel transcript gives multi-turn chat with no scripting.
- **Atomic script call — `AI.Execute`.** A Server.Class method or Script Task calls `AI.Execute(query)` (from the `T.Toolkit.LocalAI` namespace), gets a reply JSON envelope, and uses the result however it likes. No transcript, every call independent.

Both patterns share the same backend, model configuration, and enable gates. The only difference is whether the per-connection transcript cache participates.

For a complete shipping solution that exercises both patterns end-to-end — operator chat panel grounded in live plant data plus a server-side alarm-annotation script — see the LocalAI KnowledgeGraph Demo.

Default configuration

On a fresh 10.1.5 solution, Local AI ships with a loopback placeholder endpoint — `http://192.168.1.50:11434/v1/chat/completions` — replace 192.168.1.50 with the IP of the machine running your model (the model must not run on the FrameworX/TServer host). The install may seed a loopback (`localhost` / `127.0.0.1`) value, but that is a placeholder you MUST change to the model machine's IP — the model does not run on the FrameworX host. The default model is `qwen2.5:7b-instruct`, the recommended default, which expects a GPU. To run end-to-end: install Ollama on a separate GPU machine, run `ollama pull qwen2.5:7b-instruct` there, set the URL to that machine's IP, open the solution. To use a different OpenAI-compatible endpoint (including a remote or cloud LLM — see Remote and Cloud LLM Models), edit the configuration via **Solution Capabilities AI Engine Edit Configuration** (structured editor), or edit the underlying `SolutionCapabilities[LocalAI].Settings` JSON directly (see **Configuration** below).

Operator chat — the ChatRequest action

The simplest way to put a chat panel on an operator Display: three tags, one Action dynamic, one TextBox, one TextBlock. No scripting.

Step 1. Create three tags

Tag name	Type	Purpose
`Tag.Chat.UserInput`	String	Operator types into a TextBox bound here.
`Tag.Chat.ReplyJson`	**JSON** (recommended) or String	Receives the full reply envelope. Recommended type is JSON so the built-in tag methods `JsonString` and `JsonValue` can extract fields in Display Expressions with no scripting.
`Tag.Chat.LastAnswer`	String	The plain answer text. A TextBlock under the input field binds here.

Step 2. Wire the Action

On a Button (or any clickable control), add an Action dynamic with these fields:

- **Action type**: `ChatRequest`
- **Query**: `Tag.Chat.UserInput`
- **Return**: `Tag.Chat.ReplyJson`
- **Result 1**: `Tag.Chat.LastAnswer` — **Expression 1**: `@Tag.Chat.ReplyJson.JsonString("text")`

The Designer's **ChatRequest** action hides the Object editor, the HTTP-method picker, and the Force-Change checkbox — none apply when the target is the solution-wide Local AI. Only the Query, Return, and Expressions surface.

What the operator does

Types a question into the TextBox presses the button. Within ~500 ms to ~3 seconds (model dependent), `Tag.Chat.LastAnswer` populates with the reply and the TextBlock shows it. The full reply envelope (status, latency, warnings, optional tool-call trace) is on `Tag.Chat.ReplyJson` for any audit or debug panel that wants to expose it.

Tool-loop cap

The ChatRequest action dispatches at most 5 tool calls per chat turn. When the model reaches this cap, the turn returns the partial reply with `status = "truncated"`; subsequent operator turns re-enable tool-calling normally.

Multi-turn chat (default ON in 10.1.5)

By default, each Display panel keeps its own conversation history with the model — follow-up questions retain context. The transcript resets transparently when the operator on that panel logs in (shift change). To disable retained history solution-wide, clear bit `0x80` (`EnableChatHistory`) on `SolutionSettings.ModelOptions` — every chat call then behaves atomically.

Script API — AI.Execute

For server-side, single-shot LLM calls inside a Server.Class method or a Script Task, use:

```
// Synchronous — returns the full reply JSON envelope.
string reply = AI.Execute(query);

// Async overload (for native async/await scripts).
Task<string> reply = AI.ExecuteAsync(query);

// note: query is a string (your question or command to the LLM)
```

Namespace setup. The unqualified `AI.Execute` / `AI.ExecuteAsync` calls resolve when `T.Toolkit.LocalAI` is listed in the script's **Namespace Declarations**. Add it once on the Server.Class or Script Task; subsequent calls are clean. If you prefer fully-qualified names with no namespace setup, write `T.Toolkit.LocalAI.AI.Execute(query)` instead.

Legacy alias. Pre-10.1.5 scripts that call `TK.AIExecute(query)` / `TK.AIExecuteAsync(query)` still work — the flat-on-`TK` alias is retained as an `[Obsolete]` forwarder that calls `AI.Execute` internally and inherits the never-throws contract. New code uses `AI.Execute`; the alias compiles with a CS0618 warning and is hidden from IntelliSense. To migrate, change the call to `AI.Execute` and add `T.Toolkit.LocalAI` to NamespaceDeclarations — one line.

Sync or async — choose by caller context. An LLM round-trip on a local CPU model takes 0.5–10 seconds (longer for "thinking" models). Use `AI.ExecuteAsync` from any UI-bound or interactive context — Display CodeBehind, ribbon callback, animation tick — where blocking the calling thread for that long would freeze the experience. Use `AI.Execute` from Server.Class methods invoked by Script Tasks, alarm callbacks, or report generators — contexts where blocking the calling thread is acceptable. The synchronous wrapper unwraps the async call via `AsyncHelpers.RunSync`; never use raw `.Result` or `.GetAwaiter().GetResult()` on `AI.ExecuteAsync` — both deadlock under a UI SynchronizationContext. Full deep-dive: Local AI Developer Reference.

`AI.Execute` never throws. Every failure path — invalid context, model offline, network error, gate disabled — returns a well-formed reply JSON with `status = "error"` (or `"disabled"`) and an explanatory `warnings` entry. Customer scripts can rely on the reply always being parseable.

Reply shape

Error rendering macro 'code': Invalid value specified for parameter 'com.atlassian.confluence.ext.code.render.InvalidValueException'

```
{
  "text": "<the LLM's answer>",
  "status": "ok | error | disabled | truncated",
  "toolTrace": [],
  "latencyMs": 480,
  "warnings": []
}
```

Two ways to consume the reply: parse with `Newtonsoft.Json.Linq.JObject.Parse`, or assign to a tag of type JSON and use the built-in tag methods (`JsonString`, `JsonValue`).

When to use AI.Execute vs the ChatRequest action

Scenario	Use
Operator chats from a Display panel; needs follow-up questions and conversational memory.	Display **ChatRequest** action
Server.Class method needs an LLM result for a single task: rephrase, summarize, classify, translate, hypothesize.	`AI.Execute`
Alarm-event callback wants a probable-cause hypothesis attached to a tag.	`AI.Execute`
End-of-shift report Script Task wants a one-paragraph narrative summary.	`AI.Execute`

Practical examples

Three representative patterns. Each example demonstrates a use case where the LLM adds value that conventional scripting cannot — correlating multi-tag context, generating natural language, or accessing background domain knowledge.

The Server.Class containing these methods should list `T.Toolkit.LocalAI` in NamespaceDeclarations so the unqualified `AI.Execute` calls resolve.

Example 1 — Multi-tag root-cause hypothesis on an alarm

When a critical alarm fires, the operator typically scans five or six related tags to form a hypothesis about what's actually wrong. This Server.Class collects those tags automatically when the alarm activates and asks the LLM to correlate them into a probable-cause statement.

```
public void DiagnosePumpHighTemp()
{
    var snapshot = new JObject
    {
        ["alarm"]            = "Pump1.HighTempAlarm",
        ["bearingTempC"]     = (double)@Tag.Pump1.BearingTemp,
        ["motorCurrentA"]    = (double)@Tag.Pump1.MotorCurrent,
        ["dischargePressBar"]= (double)@Tag.Pump1.DischargePress,
        ["suctionPressBar"]  = (double)@Tag.Pump1.SuctionPress,
        ["flowRate_m3h"]     = (double)@Tag.Pump1.FlowRate,
        ["vibrationMmS"]     = (double)@Tag.Pump1.Vibration,
        ["ambientTempC"]     = (double)@Tag.WeatherStation.AmbientTemp,
        ["runHoursSinceMaint"] = (int)@Tag.Pump1.RunHoursSinceMaint
    };

    var query = new JObject
    {
        ["system"] = "You are a rotating-equipment reliability engineer. Given a snapshot " +
                     "of related sensor readings around a pump high-temperature alarm, " +
                     "produce ONE sentence stating the most likely root cause and ONE " +
                     "sentence with the next operator action. No preamble.",
        ["user"]    = "Diagnose this alarm.",
        ["context"] = snapshot
    };

    string reply = AI.Execute(query.ToString());
    string text  = JObject.Parse(reply).Value<string>("text") ?? "";

    @Tag.Pump1.LastDiagnosisText = text;
    @Tag.Pump1.LastDiagnosisJson = reply;
}
```

Why AI vs. without: a non-AI script could only template a fixed sentence per alarm tag. The LLM correlates eight numeric inputs against its background knowledge of pump failure modes — cavitation vs bearing failure vs blocked impeller vs cooling-water loss — and selects the explanation that fits this specific snapshot.

Example 2 — Multi-language operator alert translation

Critical alarm message is authored in English; site operators read other languages. The LLM translates while preserving technical terms (sensor IDs, units, numeric values) verbatim.

```
public void LocalizeCriticalAlarm()
{
    string englishText = @Tag.Alarm.LastCriticalMessage;
    string targetLang  = @Tag.System.LocaleForOperator;

    if (targetLang == "en" || string.IsNullOrEmpty(englishText))
    {
        @Tag.Alarm.LastCriticalMessageLocalized = englishText;
        return;
    }

    var query = new JObject
    {
        ["system"] = "You are a SCADA alarm-message translator. Translate the user's English " +
                     "alarm into the target language. Preserve tag names, sensor IDs, units, " +
                     "and numeric values verbatim. Keep it short and operator-friendly.",
        ["user"]    = englishText,
        ["context"] = new JObject { ["targetLanguage"] = targetLang }
    };

    string reply  = AI.Execute(query.ToString());
    string status = JObject.Parse(reply).Value<string>("status") ?? "error";
    string text   = JObject.Parse(reply).Value<string>("text") ?? "";

    @Tag.Alarm.LastCriticalMessageLocalized = (status == "ok") ? text : englishText;
}
```

Why AI vs. without: static translation tables don't cover the variable-content alarm message body, which has live numeric values and tag references that need to stay verbatim. The LLM applies its general translation knowledge while honouring the "preserve technical tokens" instruction.

Example 3 — End-of-shift summary

At end of shift, gather alarm events, downtime windows, and setpoint changes; LLM produces an 80–120 word manager-readable paragraph for the next operator's handoff.

```
public void GenerateShiftSummary()
{
    DateTime shiftEnd   = DateTime.Now;
    DateTime shiftStart = shiftEnd.AddHours(-8);

    JArray alarms        = QueryAlarmEvents(shiftStart, shiftEnd);
    JArray downtimes     = QueryDowntimeWindows(shiftStart, shiftEnd);
    JArray setpointEdits = QuerySetpointAuditTrail(shiftStart, shiftEnd);

    var rollup = new JObject
    {
        ["shift"]          = new JObject {
                                 ["from"] = shiftStart.ToString("o"),
                                 ["to"]   = shiftEnd.ToString("o"),
                                 ["operator"] = @Client.UserName
                               },
        ["alarms"]         = alarms,
        ["downtimes"]      = downtimes,
        ["setpointEdits"]  = setpointEdits,
        ["productionTotal"]= (double)@Tag.Plant.ShiftProduction
    };

    var query = new JObject
    {
        ["system"] = "You are a plant-operations writer. Produce ONE concise paragraph " +
                     "(80-120 words) summarizing the shift for the next operator. Cover: " +
                     "production, top alarm theme, downtime, notable setpoint changes, " +
                     "and one line on what to watch on the next shift. No bullet points.",
        ["user"]    = "Write the shift summary.",
        ["context"] = rollup
    };

    string reply = AI.Execute(query.ToString());
    string text  = JObject.Parse(reply).Value<string>("text") ?? "";

    @Tag.Shift.LastSummaryText = text;
    @Tag.Shift.LastSummaryJson = reply;
}
```

Why AI vs. without: a templated shift report is mechanical and reads as such — managers learn to skip them. The LLM connects events into a narrative that a template cannot. The cost is one LLM call per shift; the value is a report that's actually read.

Configuration

Endpoint configuration

Local AI reads its endpoint configuration from a single JSON blob on `SolutionCapabilities[LocalAI].Settings`. The shape:

Error rendering macro 'code': Invalid value specified for parameter 'com.atlassian.confluence.ext.code.render.InvalidValueException'

```
{
  "URL": "http://192.168.1.50:11434/v1/chat/completions",
  "Name": "qwen2.5:7b-instruct",
  "Authorization": "NoAuth",
  "Headers": "",
  "Info": "Recommended default model. Apache 2.0, ~4.7 GB. Replace 192.168.1.50 with the IP of the machine
running your model.",
  "TimeoutSeconds": 60
}
```

Replace 192.168.1.50 with the IP of the machine running your model (the model must not run on the FrameworX/TServer host). All six fields default sensibly — an empty or missing `Settings` resolves to the values above. Replace the URL and Name to point at any OpenAI-compatible endpoint (remote or cloud LLM, alternate local model, custom server — see Remote and Cloud LLM Models). The `Authorization` field accepts `NoAuth`, `BearerToken`, `BasicAuth`, or `CustomAuth` — the same multi-line format the WebData connector uses. Embed `/secret:<Name>` tokens to pull from the SecuritySecrets vault.

`TimeoutSeconds` is the per-call wall-clock budget in seconds (default `60`, range 30–600; values outside the range fall back to 60). A complete turn — the request plus any tool calls and the reply build — must finish inside this window, or the reply comes back with `status = "truncated"` or `"error"`. This is the authoritative budget: FrameworX imposes no shorter hidden timeout, so a configured value up to 600 seconds is honored in full. The setting is read fresh on every call, so an edit takes effect on the next request with no restart.

Running Ollama on a separate host

Local AI works equally well when Ollama runs on a different machine from FrameworX — and this is the recommended arrangement. Do not run the model on the FrameworX/TServer host — it competes with the runtime for CPU; use a separate GPU machine or a remote/cloud endpoint. For remote and cloud endpoints specifically, see Remote and Cloud LLM Models. Typical reasons to split the deployment:

- **GPU-equipped Ollama box.** Keep the SCADA / Designer workstation on its own hardware; concentrate the model serving on a GPU machine where 7B (or 32B) responses stay sub-second.
- **Lifecycle separation.** Production deployments often want the FX runtime and the model server on separate boxes so they can be upgraded, restarted, or scaled independently.
- **Shared model server.** One Ollama host serves multiple FrameworX solutions or sites — one model pull, one cache, multiple consumers.

On the Ollama host machine. By default Ollama binds `localhost` only. To accept remote connections, set `OLLAMA_HOST=0.0.0.0:11434` in the system environment, restart Ollama, then open inbound TCP 11434 in the host firewall.

On the FrameworX side. Edit `SolutionCapabilities[LocalAI].Settings` and set the `URL` field to `http://192.168.1.50:11434/v1/chat/completions` — replace 192.168.1.50 with the IP of the machine running your model (the model must not run on the FrameworX/TServer host). No other field needs to change for a trusted LAN deployment.

Network considerations. Ollama has no built-in authentication. For any deployment beyond a trusted LAN, restrict the firewall rule on the Ollama host to the FX server's IP, OR front port 11434 with a reverse proxy that adds an API key — then set the FX `Authorization` field to `BearerToken` with that key. Do not expose port 11434 directly to an untrusted network.

Latency. The first call after the model loads into RAM is ~10–30 seconds depending on model size; subsequent calls are typically sub-second on the same model. Network latency between FX and Ollama adds a few milliseconds on a LAN — negligible compared to inference time.

The First Install Walkthrough's **Running the model on a different host** section carries the equivalent procedure with script-level detail.

Enable bits — `SolutionSettings.ModelOptions`

Local AI shares the same `ModelOptions` integer surface that gates the AI Runtime Connector. Each bit is independently set:

Bit	Name	Effect when ON
`0x02`	EnableRuntimeMCP (master)	Master enable for all AI features. When OFF, ChatRequest and `AI.Execute` return `status = "disabled"`.
`0x04`	EnableUnsTools	The LLM can read tag values and browse the namespace when it decides to use those tools.
`0x08`	EnableAlarmTools	The LLM can read active alarms and query the alarm history.
`0x10`	EnableHistorianTools	The LLM can query historian time-series data.
`0x20`	EnableCustomTools	The LLM can call solution-authored MCP Tool methods (10.1.5+).
`0x40`	EnableDesignerMCP	Reserved for the AI Designer connector. Do not reuse for Local AI features.
`0x80`	EnableChatHistory	Per-Display-panel transcript cache participates in ChatRequest calls. Default **ON**. `AI.Execute` always bypasses the cache regardless of this bit.

What Local AI does NOT do

- It does not stream replies token-by-token. Each call returns one complete envelope when the model finishes.
- It does not run on a connected client / Display directly. All LLM calls execute server-side on TServer.
- It does not throw on failure. Every error path returns a parseable reply envelope with `status` set to `error`, `disabled`, or `truncated`.
- It does not retry on transient failure. A failed call returns immediately with the error reply; the customer's calling code decides whether to retry.

In this section...