Local AI

FrameworX Local AI is the platform's built-in, on-device 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+

Local AI ships pre-configured to use a local Ollama with `qwen2.5:7b-instruct` by default (Apache 2.0, ~4.7 GB). One-time ~5-minute scripted install — see the **First Install Walkthrough** child page for details.

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 — `TK.AIExecute`.** A Server.Class method or Script Task calls `TK.AIExecute(query)`, gets a SPEC §14.2 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.

Default configuration

On a fresh 10.1.5 solution, Local AI is configured to talk to a local Ollama at `http://localhost:11434/v1/chat/completions` using `qwen2.5:7b-instruct`. To run end-to-end out of the box: install Ollama, run `ollama pull qwen2.5:7b-instruct`, open the solution — chat works immediately. To use a different model or a different OpenAI-compatible endpoint, edit the `SolutionSettings.ModelSettings` JSON config (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.

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 — TK.AIExecute

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 = TK.AIExecute(query);

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

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

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 `TK.AIExecuteAsync` 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 `TK.AIExecute` 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 `TK.AIExecuteAsync` — both deadlock under a UI SynchronizationContext. Full deep-dive: Local AI Developer Reference.

`TK.AIExecute` 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 TK.AIExecute 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.	`TK.AIExecute`
Alarm-event callback wants a probable-cause hypothesis attached to a tag.	`TK.AIExecute`
End-of-shift report Script Task wants a one-paragraph narrative summary.	`TK.AIExecute`

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.

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 = TK.AIExecute(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  = TK.AIExecute(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 = TK.AIExecute(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 the existing `SolutionSettings.ModelSettings` column. The shape:

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

```
{
  "URL": "http://localhost:11434/v1/chat/completions",
  "Name": "qwen2.5:7b-instruct",
  "Authorization": "NoAuth",
  "Headers": "",
  "Info": "Default local model. Apache 2.0, ~4.7 GB."
}
```

All five fields default sensibly — an empty or missing `ModelSettings` column resolves to the values above. Replace the URL and Name to point at any OpenAI-compatible endpoint (cloud LLM, alternate local model, custom server). 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.

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 TK.AIExecute 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 (10.1.5.1+).
`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**. `TK.AIExecute` 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...