

# Local AI Developer Reference

Advanced developer reference for programming against Local AI in FrameworkX 10.1.5: `TK.AIExecute` deep semantics, hook attachment via reflection, custom MCP tool exposure, audit and redaction patterns, error envelope decision matrix, and worked code for non-trivial integrations.

[Technical Reference](#) [Programming and APIs Reference](#) Local AI Developer Reference

Version 10.1.5+

## Audience and scope

This page is for solution developers and integrators who need to go beyond the configuration UI and reference docs — attaching audit hooks, exposing custom tools to the LLM, building diagnostic instrumentation around chat turns, or wrapping `TK.AIExecute` in higher-level abstractions specific to their solution.

## The two surfaces, side by side

Aspect	<code>TK.AIExecute</code> (atomic)	<code>ChatRequest</code> action (cached)
Caller	Server-domain script	Display dynamic
Stateful	No	Yes (per-Display-panel transcript)
Tools the LLM can call	None	<code>runtime_* set</code> + custom <code>Script.Class</code> methods
Hooks raised	None	<code>OnBeforeChat</code> , <code>OnAfterChatReply</code>
Master gates	<code>ModelEnabled</code> only	<code>ModelEnabled</code> + <code>ModelOptions</code> bit <code>0x02</code>
Wall-clock budget	60 s for the single POST	60 s covering the whole tool-dispatch loop
Per-turn tool dispatch cap	n/a (no tools)	<code>MaxToolDispatchesPerTurn = 5</code>
Reply envelope	Same shape; <code>toolTrace</code> always <code>[]</code>	Same shape; <code>toolTrace</code> populated when tools dispatched

## `TK.AIExecute` deep semantics

### Sync vs async — pick the right wrapper

```
// Synchronous wrapper - blocks the caller until the reply arrives.
string reply = TK.AIExecute(string query);

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

The synchronous wrapper unwraps the async call via the platform helper `T.Library.tRPC.AsyncHelpers.RunSync`. Do NOT use raw `.Result` or `.GetAwaiter().GetResult()` on `TK.AIExecuteAsync`:

- `.Result` deadlocks under a UI `SynchronizationContext` (Display `CodeBehind`, certain Designer-side scripts).
- `.GetAwaiter().GetResult()` swallows the unwrap and surfaces an `AggregateException` instead of the inner exception, hiding the real cause.

The platform helper handles `SynchronizationContext` correctly and unwraps `AggregateException` to the inner exception. The same helper is the right pattern anywhere in FrameworkX where sync-over-async is needed.

### The ambient `ObjectServer`

`TK.AIExecute` resolves its `ObjectServer` from `TK.ObjServer` — the ambient `ObjectServer` in the calling code path. In a Server-domain script context this is the server-side `ObjectServer`; in a Display `CodeBehind` context it is the client-side instance.

The platform does not block client-side callers with a runtime gate; it relies on platform safety nets (secret resolver short-circuit, defensive defaults, master kill-switch) to surface a normal HTTP-error envelope rather than silently transmit unauthenticated traffic. See [Local AI Architecture Reference](#) § "Why no runtime view-client gate" for the full rationale.

### Query format — plain vs structured

The `query` parameter accepts two shapes:

Shape	Example	When to use
Plain text string	"Translate to French: Pump 1 is offline."	Trivial single-shot tasks where no system prompt or context envelope is needed.
Structured JSON object	{ "system": "...", "user": "...", "context": { ... } }	Anything that needs a system prompt to set persona / output constraints, or that needs structured context the LLM should consider.

Structured fields:

Field	Required	Notes
user	Yes (when using structured form)	The user-role message — the actual question or instruction.
system	No	The system-role message — sets persona, voice, output constraints.
context	No	Arbitrary structured data serialized into the LLM prompt for grounding.
metadata	No	Reserved for caller-side annotation; the platform does not currently consume it.

## Reply envelope — parsing pattern

The reply is always parseable JSON. Idiomatic dispatch:

```
string replyJson = TK.AIExecute(query);
JsonObject reply = JObject.Parse(replyJson);

string status = reply.Value<string>("status"); // never null
long ms = reply.Value<long>("latencyMs"); // always parseable
string text = reply.Value<string>("text") ?? "";

switch (status)
{
    case "ok":
        ConsumeAnswer(text);
        break;
    case "disabled":
        // Master kill-switch is off. warnings[] names the gate.
        LogDisabled((JArray)reply["warnings"]);
        break;
    case "truncated":
        // Wall-clock budget exceeded. text may carry a partial answer.
        ConsumePartial(text, (JArray)reply["warnings"]);
        break;
    default: // "error"
        LogError((JArray)reply["warnings"], ms);
        break;
}
```

## Failure modes — exhaustive

Trigger	status	Representative warnings entry	latencyMs
ModelEnabled = false	disabled	"Local AI master kill-switch (ModelEnabled) is off."	0
Query JSON malformed	error	"Invalid query JSON: <parser message>"	elapsed
Structured form missing user	error	"Query missing required field 'user'."	elapsed
URL empty after defaults	error	"AI endpoint URL is empty (LocalAIDefaults misconfigured?)."	elapsed
Wall-clock budget exceeded	truncated	"LLM POST wall-clock budget (60s) exceeded."	60000
HTTP non-2xx response	error	"LLM endpoint HTTP error: 401 Unauthorized"	elapsed
Network failure	error	"LLM endpoint HTTP error: Connection refused"	elapsed
Other unhandled exception	error	"<exception type>: <message>"	elapsed

TK.AIExecute never throws — every failure path lands here.

## Hook attachment via reflection (cached path)

The cached `ChatRequest` path raises two hooks per turn:

Hook	Fires	Receives / returns	Typical use
<code>OnBeforeChat</code>	Before the LLM POST	Receives the query JSON; returns possibly-mutated query JSON	PII redaction, query rewriting, audit, rate-limiting
<code>OnAfterChatReply</code>	After the reply envelope is built, before return	Receives the reply JSON; returns possibly-mutated reply JSON	Post-processing, additional audit, in-place rewriting

## Hook semantics

- **Multicast.** Multiple handlers may attach; the chain runs sequentially.
- **Async.** Each handler is `Func<string, Task<string>>`; awaited in order.
- **Mutation.** Returning modified JSON is honoured; subsequent handlers see the modified version. Returning `null` is treated as "no mutation".
- **Error isolation.** A throwing handler is caught; `warnings[]` records "`<HookName> handler '<MethodName>' threw: <message>`"; the chain proceeds with the un-mutated input.
- **No status flip.** Hook failures do NOT change the envelope's `status` from `ok` to `error` — the LLM call still proceeds. `status="error"` is reserved for chain-aborting failures (HTTP error, malformed query, etc.).
- **Cached path only.** Atomic `TK.AIExecute` calls do NOT raise these hooks — chat-attached behaviour does not apply.

## Attaching a handler

In 10.1.5, attachment uses reflection from a Server-domain script. The first-class binding surface (`@Solution.AI.OnBeforeChat += handler`) is post-10.1.5 work.

```
using System;
using System.Reflection;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;

public class LocalAIHookSetup
{
    // Call this once at solution startup – typically from a ServerStartup task
    // or a Server.Class method invoked during initialization.
    public static void AttachAll()
    {
        AttachOnBeforeChat(RedactPII);
        AttachOnAfterChatReply(LogReplyToAudit);
    }

    // ---- handler implementations ----

    static async Task<string> RedactPII(string queryJson)
    {
        try
        {
            var query = JObject.Parse(queryJson);
            string user = query.Value<string>("user") ?? "";

            // Strip 9-digit sequences that look like SSNs.
            user = System.Text.RegularExpressions.Regex.Replace(
                user, @"\"b\d{3}-?\d{2}-?\d{4}\"b", "[REDACTED]");

            query["user"] = user;
            return query.ToString();
        }
        catch
        {
            // Returning the input unchanged on failure is safe; the chain
            // will continue with the un-mutated input either way.
            return queryJson;
        }
    }

    static async Task<string> LogReplyToAudit(string replyJson)
    {
        try
        {
```

```

var reply = JObject.Parse(replyJson);
string status = reply.Value<string>("status") ?? "";
long ms = reply.Value<long>("latencyMs");
string text = reply.Value<string>("text") ?? "";

// Append to a Dataset, write to a tag, or emit to a custom log.
// (See your Datasets module configuration for the right table.)
// TK.WriteToDataset("AIChatAudit", new { status, ms, text, when = DateTime.UtcNow });
return replyJson;
}
catch
{
return replyJson;
}
}

// ---- reflection plumbing ----

static void AttachOnBeforeChat(Func<string, Task<string>> handler)
=> AttachHook("OnBeforeChat", handler);

static void AttachOnAfterChatReply(Func<string, Task<string>> handler)
=> AttachHook("OnAfterChatReply", handler);

static void AttachHook(string eventName, Func<string, Task<string>> handler)
{
var aiType = T.Library.TAssembly.GetType(null, "T.Toolkit.LocalAI.LocalAIService");
if (aiType == null)
throw new InvalidOperationException("Local AI service type not found.");

var getMethod = aiType.GetMethod(
"Get",
BindingFlags.NonPublic | BindingFlags.Static,
null,
new[] { typeof(ObjectServer) },
null);

var instance = getMethod.Invoke(null, new object[] { TK.ObjServer });

var eventInfo = aiType.GetEvent(
eventName,
BindingFlags.NonPublic | BindingFlags.Instance);

eventInfo.AddEventHandler(instance, handler);
}
}

```

## Detaching a handler

Symmetric: keep the `Func<string, Task<string>>` reference around and call `eventInfo.RemoveEventHandler(instance, handler)` when the handler should stop firing. Handlers attached via lambda without a captured reference cannot be detached individually.

## Lifetime considerations

- Handlers persist for the lifetime of the TServer process — there is no automatic detach on solution reload.
- If `AttachAll` is called more than once (multiple `ServerStartup` invocations on certain reload paths), each attachment adds another handler to the chain. To avoid double-firing, gate with a static `bool _attached` or detach previously-attached handlers first.
- Throwing from a handler is safe (caught, recorded in warnings, chain continues), but a slow handler is not — the platform awaits each handler sequentially and counts the elapsed time against the 60 s wall-clock budget. Keep handlers fast or move slow work onto a fire-and-forget task.

## Custom MCP tools — exposing solution methods to the LLM

The cached `ChatRequest` path can dispatch solution-authored tools when `SolutionSettings.ModelOptions bit 0x20 (EnableCustomTools)` is set in addition to the master bit `0x02`.

## Authoring a custom tool

A custom tool is a Script.Class method tagged with the platform's MCP-tool attribute. The class registers itself at solution startup; the cached path's tool-catalog assembler discovers tagged methods and exposes them to the LLM as OpenAI-compatible `tools[]` entries.

```
// In a Script.Class, e.g. "PlantTools":
public class PlantTools
{
    // Each method exposed to the LLM is a public method on a Script.Class
    // instance. The platform's tool-catalog discovery enumerates these
    // methods and builds OpenAI-compatible function descriptors from
    // their parameter signatures and XML doc comments.

    /// <summary>
    /// Returns the current production rate for a given line, in units/hour.
    /// </summary>
    /// <param name="lineId">Production line identifier (e.g. "Line1").</param>
    public double GetProductionRate(string lineId)
    {
        // Read from the live solution. Toolkit access works because the
        // method runs in TServer under a server-side ObjectServer.
        return (double)TK.GetTagValue($"Tag.{lineId}.ProductionRate");
    }

    /// <summary>
    /// Returns the count of active alarms in the named area.
    /// </summary>
    /// <param name="area">Plant area name (e.g. "BoilerHouse").</param>
    public int GetActiveAlarmCount(string area)
    {
        // Replace with the right tag / dataset query for your solution.
        return TK.QueryActiveAlarms(area).Count;
    }
}
```

## How the LLM sees the tools

At chat-turn assembly, the platform inspects the registered Script.Class methods and produces function descriptors of the form:

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

```
{
  "type": "function",
  "function": {
    "name": "PlantTools_GetProductionRate",
    "description": "Returns the current production rate for a given line, in units/hour.",
    "parameters": {
      "type": "object",
      "properties": {
        "lineId": { "type": "string", "description": "Production line identifier (e.g. Line1)." }
      },
      "required": [ "lineId" ]
    }
  }
}
```

Argument descriptions are pulled from the XML `<param>` tags; method descriptions from `<summary>`. Write these as if they were API docs — the LLM reads them to decide when to call.

## Dispatch trace

Every tool dispatch appears in the reply envelope's `toolTrace[]` array (see [Local AI Reply Envelope Schema](#)):

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

```
{
  "name": "PlantTools_GetProductionRate",
  "args": { "lineId": "Line1" },
  "result": 245.7,
  "status": "ok",
  "timestamp": "2026-04-26T14:32:15.123Z",
  "elapsedMs": 4
}
```

## Custom tool guidelines

- **Pure functions are easiest.** A method that reads tags and returns a value is straightforward to expose. Methods with side effects (writing tags, sending emails) demand more thought — the LLM will call the method when it judges the call useful, which may not match operator expectations.
- **Parameter types should be JSON-friendly.** Strings, numbers, booleans, arrays, and POCOs that serialize cleanly. Avoid passing platform-internal types.
- **Return types should be small and structured.** A few hundred bytes of structured JSON is ideal. Returning a 10 MB payload bloats the LLM context for the rest of the turn and may push the wall-clock budget.
- **Document well.** The LLM picks tools based on the descriptions. Vague descriptions produce vague tool selection.
- **Mind the per-turn cap.** The cached path enforces `MaxToolDispatchesPerTurn = 5`. If a chat turn realistically needs more than 5 dispatches, consider consolidating multiple reads into a single coarser-grained tool that returns a structured payload.

## Audit logging — capturing every chat turn

Pattern: attach a single `OnAfterChatReply` hook that writes the turn outcome to a dataset row, a tag, or both. Recommended schema:

Column	Source
<code>WhenUtc</code>	<code>DateTime.UtcNow</code> in the hook
<code>Status</code>	<code>reply.Value&lt;string&gt;("status")</code>
<code>LatencyMs</code>	<code>reply.Value&lt;long&gt;("latencyMs")</code>
<code>AnswerExcerpt</code>	First N chars of <code>reply.Value&lt;string&gt;("text")</code>
<code>Warnings</code>	<code>reply["warnings"]</code> as a comma-joined string
<code>ToolCount</code>	<code>((JArray)reply["toolTrace"]).Count</code>

For the matching `OnBeforeChat` half (capturing the query that produced the reply), correlate by inserting an opaque `turnId` into the query's `metaData` field and reading it back in the after-hook.

## Redaction — stripping content from queries before they leave the platform

Pattern: attach a single `OnBeforeChat` hook that walks the structured query, applies redaction rules to the `user` field (and `context` if needed), and returns the modified JSON.

Considerations:

- **Apply to structured fields only.** If the caller passed plain text, parse-and-rewrap to structured form, redact, and re-stringify.
- **Failure-safe defaults.** If parsing fails, return the input unchanged — the platform's chain semantics ensure the un-mutated input flows through.
- **Stack with audit.** The audit hook (above) sees the redacted query, not the original. If you need both, write the original to audit before the redaction hook runs (handler order is registration order).

## Performance characteristics

Phase	Typical latency on a CPU	Notes
Cold load (model into RAM)	~15 s	First call after TServer startup or after the model's keep-alive window expires
Warm inference (per LLM POST)	~3–15 s	CPU-only; ~0.5–2.5 s on GPU; depends on model and reply length
Per tool dispatch round-trip (cached path)	~1–3 s	LLM thinks in-process tool exec (sub-millisecond) LLM thinks again
Hook handler	Add to wall-clock budget	Each handler awaited sequentially; counts against 60 s budget
Reply parsing in caller	< 1 ms	Single <code>JObject.Parse</code>

## Why the per-turn cap is 5

The cached path enforces `MaxToolDispatchesPerTurn = 5` in 10.1.5. The cap is intentionally conservative for CPU-only deployments — common on industrial PCs and edge gateways without GPUs — where each LLM POST can take 5–15 seconds. A turn that hits the cap consumes up to 6 LLM POSTs (5 dispatches plus the terminal answer), which on the slow end of CPU latency lands inside the 60-second wall-clock budget. On GPU-class hardware (~0.5–2.5 s per POST) the cap is not the binding constraint — the wall-clock budget is. Realistic SCADA queries (single-tag read, alarm count, multi-tag diagnosis) rarely exceed 3–4 dispatches; 5 covers the typical operator interaction comfortably while preventing runaway loops.

For the cold-load cost, set `OLLAMA_KEEP_ALIVE=24h` in the environment of the Ollama host so the model stays resident.

## Threading and ordering

- **Cached path** runs on the `TCPServer` dispatch thread for the inbound `ServiceClient` call. Hook handlers and tool dispatches execute synchronously on the same thread (each `await` chained sequentially).
- **Atomic path** runs on the calling script's thread. The sync wrapper unwraps async via `AsyncHelpers.RunSync`; the underlying async chain runs on whichever thread pool slot the platform helper assigns.
- **Multiple concurrent chat turns** from different Display panels are isolated by transcript cache key (`ClientInfo.Guid`). The Local AI service singleton is per-`ObjectServer`, so multiple turns share the singleton but operate on distinct cache entries and distinct outbound HTTP contexts.
- **Hook handler ordering** is registration order. If two handlers depend on each other (e.g., audit must see the original before redaction), register them in the order they should fire.

## Worked example: chat-turn audit dataset

Wire `OnAfterChatReply` to append a row to a dataset called `AIChatAudit` on every cached turn. Datasets module configuration is out of scope for this page; the hook implementation:

```
static async Task<string> AppendChatAudit(string replyJson)
{
    try
    {
        var reply = JObject.Parse(replyJson);
        var row = new JObject
        {
            ["WhenUtc"]           = DateTime.UtcNow.ToString("o"),
            ["Status"]            = reply.Value<string>("status"),
            ["LatencyMs"]         = reply.Value<long>("latencyMs"),
            ["AnswerExcerpt"]     = Truncate(reply.Value<string>("text") ?? "", 256),
            ["Warnings"]          = string.Join("; ",
                ((JArray)reply["warnings"])
                    .Values<string>()),
            ["ToolCount"]         = ((JArray)reply["toolTrace"]).Count,
            ["UserName"]          = TK.UserName ?? "",
            ["ClientGuid"]        = TK.ClientGuid ?? ""
        };

        // Replace with your dataset insert call.
        // TK.DatasetInsert("AIChatAudit", row);
    }
    catch
    {
        // Audit failures must not break chat - swallow and continue.
    }
    return replyJson;
}

static string Truncate(string s, int max)
    => s.Length <= max ? s : s.Substring(0, max - 1) + "...";
```

## Worked example: rate-limit per operator

Reject queries exceeding N per minute per operator by returning a query JSON with a sentinel `user` message that produces a fast deterministic reply (or by returning unmutated and side-channel-rejecting via a separate mechanism).

```

using System.Collections.Concurrent;

static readonly ConcurrentDictionary<string, (DateTime windowStart, int count)> _rl
    = new ConcurrentDictionary<string, (DateTime, int)>();

const int MaxPerMinute = 30;

static async Task<string> RateLimit(string queryJson)
{
    string user = TK.UserName ?? "anon";
    var now = DateTime.UtcNow;

    var (windowStart, count) = _rl.GetOrAdd(user, _ => (now, 0));
    if ((now - windowStart).TotalSeconds >= 60)
    {
        windowStart = now;
        count = 0;
    }
    count++;
    _rl[user] = (windowStart, count);

    if (count > MaxPerMinute)
    {
        // Rewrite the query to a no-op that the LLM will answer briefly.
        // The reply envelope still comes back as status="ok" - log the
        // limit-hit elsewhere if you need a hard reject signal.
        var capped = new JObject
        {
            ["system"] = "Reply with the exact text: 'Rate limit reached. Please wait a moment.'",
            ["user"] = "ack"
        };
        return capped.ToString();
    }
    return queryJson;
}

```

For a hard-reject pattern (returning `status="error"` directly without an LLM round-trip), the post-10.1.5 first-class binding surface will expose a "short-circuit" signal. In 10.1.5, the LLM round-trip happens regardless — the rewrite above keeps the round-trip cheap and predictable.

## Anti-patterns

- **Don't poll the reply envelope.** `TK.AIExecute` is sync; it returns when the reply is ready. There is no "result tag" for an async result that needs polling.
- **Don't ignore status.** Treating text as authoritative without checking status means a disabled or error envelope (with empty text) silently produces an empty UI field. Always dispatch on status.
- **Don't catch and re-throw inside hooks.** Hook errors are caught by the platform and recorded in `warnings[]`. Re-throwing achieves nothing the platform doesn't already do, and sometimes loses the original exception.
- **Don't depend on hook ordering across solution loads.** If hooks A and B both register at startup, they run in registration order — but if a future change reorders the registration, behaviour changes silently. Make hooks order-independent where possible.
- **Don't block on slow I/O inside a hook.** The platform awaits each handler sequentially against the 60 s budget. Long-running work belongs in a fire-and-forget task spawned from the hook, not in the hook itself.
- **Don't pass huge context blobs.** A 50 KB `context` object inflates the LLM prompt for every turn. Trim to the minimum the LLM needs to answer.
- **Don't expose tools that decompose work the LLM would otherwise batch.** Five fine-grained "GetTagX", "GetTagY", "GetTagZ" tools force the LLM to make 3 dispatches when one "GetPlantSnapshot" tool returning all three values would do it in one. Coarse-grained tools fit the per-turn cap better.

## Cross-references

- [Local AI](#) — capability page with quick-start and worked examples.
- [Local AI Architecture Reference](#) — internal architecture, code organization, gate semantics.
- [Local AI Configuration](#) — endpoint, master enable, bitmask.
- [ChatRequest Action Reference](#) — operator-chat surface.
- [TK.AIExecute API Reference](#) — atomic script API surface.
- [Local AI Reply Envelope Schema](#) — uniform reply schema.
- [SecuritySecrets Authentication for Local AI](#) — credential management.
- [AI Runtime Developer Reference](#) — peer reference for the AI Runtime Connector (external AI clients connecting to TServer).

In this section...