# ChatRequest Action Reference

Reference for the `ChatRequest` Display action — the operator-facing chat surface for Local AI. Covers configuration, transcript persistence, hook attachment, and tool dispatch.



## What it is

`ChatRequest` is a Display Action type — the same dynamic kind as `Click`, `HTTPRequest`, `SetValue`. It runs when an operator interacts with a Display control (typically a Button) that has the action attached. The action sends the operator's typed query to the solution's Local AI endpoint, holds the per-Display-panel transcript across follow-up turns, dispatches tool calls the LLM may request, and writes the reply back to a tag the Display reads.

It is portable across Display targets — WPF (rich client) and OpenSilver (HTML5/browser) render and execute it identically.

## Configuration on the Action

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

| Field | Setting |
|---|---|
| Action type | `ChatRequest` |
| Query | Tag whose value is the operator's typed query (typically a String tag bound to a TextBox). |
| Return | Tag that receives the full reply envelope JSON. Recommended type: **JSON**, so the built-in tag-method surface (`JsonString`, `JsonValue`) can extract fields in Display Expressions without scripting. |
| Result 1, Result 2, ... (optional) | Tags whose values are computed from the reply via Expressions. Typically Result 1 is bound to a String tag holding the plain-text answer, with Expression `@Tag.Reply.JsonString("text")`. |

The Action editor hides fields that don't apply to `ChatRequest` (the HTTP method picker, the Object selector, the Force-Change checkbox). Only Query, Return, and the Result/Expression rows surface.

## The minimum operator chat panel

Three tags, one Action, two controls. No scripting.

| Tag | Type | Wired to |
|---|---|---|
| `Tag.Chat.UserInput` | String | TextBox **write** binding |
| `Tag.Chat.ReplyJson` | JSON | Action **Return** field |
| `Tag.Chat.AnswerText` | String | Result 1 with Expression `@Tag.Chat.ReplyJson.JsonString("text")`; TextBlock **read** binding |

Operator types into the TextBox, presses the Button. Within a few hundred milliseconds to a few seconds (model-dependent), the answer appears in the TextBlock. The full reply envelope (status, tool trace, latency, warnings) sits on the JSON tag and can be surfaced for diagnostics or hidden.

## Per-Display-panel transcript (multi-turn chat)

Each Display panel keeps its own conversation history with the model. Follow-up questions on the same panel retain context — the model sees the full transcript and can refer back to earlier turns. Default **ON** for new 10.1.5 solutions.

### Cache identity

The transcript is keyed by the panel's `ClientInfo.Guid` — an internal per-TCP-connection identifier the platform assigns when a client connects. Two operators chatting on two panels get two independent transcripts. The same operator with two browser tabs gets two transcripts. The runtime `Client.*` namespace does not expose the Guid; it's an internal cache key.

### Login reset

When the calling user changes on the same panel (shift change, RBAC re-login), the transcript clears transparently before the next turn. Operator A's chat history is not visible to Operator B. The reset is *lazy* — it happens on the next chat turn after the user change, not on the user change itself.

### Disabling per-solution

To disable transcript persistence solution-wide, clear bit `0x80` (`EnableChatHistory`) on `SolutionSettings.ModelOptions`. Every `ChatRequest` call then behaves atomically — the LLM sees only the current turn, with no prior context. The atomic `TK.AIExecute` script API always bypasses the cache regardless of this bit.

## Tool dispatch (the LLM acting on the solution)

When tool-category bits are enabled in `SolutionSettings.ModelOptions`, the LLM may decide during a chat turn to call platform tools that read tag values, browse the namespace, query alarms, or query historian data. The platform dispatches these in-process against the running solution and feeds the results back to the LLM, which composes a final answer. The full sequence — every tool name, arguments, result, and timing — appears in the reply envelope's `toolTrace[]` array.

### Tool surface controlled by ModelOptions

| Bit | Tools enabled |
|---|---|
| `0x02` `EnableRuntimeMCP` (master) | Required for ANY tool to be exposed. Off  no tools, even if sub-bits are on. |
| `0x04` `EnableUnsTools` | Read tag values, browse the UNS, search the namespace, get object context. |
| `0x08` `EnableAlarmTools` | Read active alarms, query alarm history. |
| `0x10` `EnableHistorianTools` | Query historian time-series data. |
| `0x20` `EnableCustomTools` | Call solution-authored MCP Tool class methods. |

See Local AI Configuration for the complete bit reference.

### Dispatch loop bounds

To prevent runaway tool loops, the platform caps each chat turn at:

- **Up to 5 tool dispatches per turn** (`MaxToolDispatchesPerTurn = 5`). Conservative cap chosen for the wall-clock budget below: at typical CPU-LLM latencies (3–15 s per round-trip), 5 dispatches plus the terminal POST keep the turn inside 60 seconds. 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.
- **60-second total wall-clock budget** for the entire turn (LLM POSTs + all tool dispatches combined).

When the cap trips, the platform gives the LLM one final round (no new tools) to compose a terminal reply from the tools it has seen, and returns `status="truncated"` if the LLM still does not produce a terminal answer.

## Hooks (OnBeforeChat / OnAfterChatReply)

Server-domain code can attach handlers to two hooks that fire on every `ChatRequest` turn:

- **`OnBeforeChat`** — fires before the LLM POST. Receives the query JSON; may return modified query JSON. Use for redaction, audit-logging, rate-limiting, or rewriting the query before it leaves the platform.
- **`OnAfterChatReply`** — fires after the reply envelope is built, before it returns to the caller. Receives the reply JSON; may return modified reply JSON. Use for post-processing, additional audit, or in-place rewriting.

Both hooks are multicast — multiple handlers may attach. A throwing handler is caught and logged into the reply's `warnings[]` with the handler's identity; the chain continues with the un-mutated input. Hook failures do NOT flip `status` to `error` — the LLM call still proceeds.

### Hooks fire on ChatRequest only

Atomic `TK.AIExecute` calls do NOT raise these hooks. Hooks are scoped to the operator-chat surface specifically — a `TK.AIExecute` call from an alarm-describer or a report generator should not fire chat-redaction or chat-audit handlers, because the contexts and policies are different.

### Attaching a handler from a Server.Class

In FrameworX 10.1.5, hook attachment uses reflection from a Server-domain script (typically a Server.Class method invoked at solution startup). A first-class binding surface (`@Solution.AI.OnBeforeChat += handler`) is post-10.1.5 work.

```
// In a Server.Class method invoked at solution startup:
public void AttachChatHooks()
{
    var aiType    = T.Library.TAssembly.GetType(null, "T.Toolkit.LocalAI.LocalAIService");
    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("OnBeforeChat",
                       BindingFlags.NonPublic | BindingFlags.Instance);

    Func<string, Task<string>> handler = async json =>
    {
        // Customer logic: read query JSON, mutate, return new JSON.
        // Throwing is safe — the chain continues with the un-mutated input.
        return json;
    };

    eventInfo.AddEventHandler(instance, handler);
}
```

The handler must execute in a server-domain context — `TK.ObjServer` must be the server-side ObjectServer. Display CodeBehind cannot attach handlers (those scripts execute at the client).

## Master gates

`ChatRequest` applies the gates in this fixed order on every call:

1. `SolutionSettings.ModelEnabled = true` — kill-switch must be ON. Off returns `status="disabled"`.
2. `SolutionSettings.ModelOptions` bit `0x02` (`EnableRuntimeMCP`) — master tool-surface bit. Off returns `status="disabled"`.
3. Per-category sub-bits — AND-ed against the master bit when assembling the tool catalog the LLM sees.

See Local AI Configuration for full gate semantics.

## Reply envelope

Same shape as `TK.AIExecute` — see Local AI Reply Envelope Schema. Briefly:

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

```
{
  "text":      "<the LLM answer text or '' on non-ok status>",
  "status":    "ok | error | disabled | truncated",
  "toolTrace": [ /* zero or more dispatched tools */ ],
  "latencyMs": 1247,
  "warnings":  [ /* zero or more diagnostic strings */ ]
}
```

## Latency expectations

| Phase | Typical latency |
|---|---|
| First call after startup or after the model's keep-alive expires | ~15 seconds (model loads from disk into RAM) |
| Subsequent calls (model resident) | ~500 ms on a typical CPU; faster with a GPU |
| Tool-equipped chat turn | Add ~1–3 seconds per dispatched tool round-trip; the LLM may dispatch zero, one, or several tools per turn |

For the cold-start cost, the recommended workaround is setting `OLLAMA_KEEP_ALIVE=24h` in the environment of the Ollama host so the model stays resident.

## Worked example — chat panel with diagnostics

An operator chat panel that exposes the answer text plus a small "details" line showing latency and status:

| Tag | Type | Wired to |
|---|---|---|
| Tag.Chat.UserInput | String | TextBox write |
| Tag.Chat.ReplyJson | JSON | Action Return |
| Tag.Chat.AnswerText | String | Result 1, Expression @Tag.Chat.ReplyJson.JsonString("text"); TextBlock |
| Tag.Chat.LastStatus | String | Result 2, Expression @Tag.Chat.ReplyJson.JsonString("status"); small label |
| Tag.Chat.LastLatencyMs | Long | Result 3, Expression @Tag.Chat.ReplyJson.JsonValue<long>("latencyMs"); small label |

No scripting needed; everything is wired in the Action editor and Display Expressions.

## What ChatRequest does NOT do

- Does not stream replies token-by-token. Each call returns one complete envelope when the model finishes.
- Does not run the LLM at the client. All HTTP traffic to the LLM happens server-side; the client only carries the operator's query and receives the reply via the existing platform tunnel.
- Does not throw on failure. Every error path returns a parseable reply envelope with status set to error, disabled, or truncated.
- Does not retry on transient failure. A failed call returns immediately with the error reply; the customer's calling code (or operator) decides whether to retry.
- Does not run on Display CodeBehind handlers. The action is fired by Display dynamics — Click, Touch, Timer — not by code-behind methods.

## See also


- Local AI — the parent reference, includes a step-by-step quick-start.
- Local AI Configuration — endpoint and bitmask configuration.
- TK.AIExecute API Reference — the atomic script-side counterpart.
- Local AI Reply Envelope Schema — full reply schema.


---

### In this section...