TK.AIExecute API Reference

Reference for `TK.AIExecute` — the server-side script API that calls Local AI atomically (one-shot, stateless, no transcript, no tool dispatch).

AI Integration Local AI TK.AIExecute API Reference

What it is

`TK.AIExecute` is a static method on the FrameworX `TK` Toolkit class. A server-domain script calls it with a query string and gets back a SPEC reply envelope JSON. Every call is an independent one-shot LLM round-trip — no transcript, no caching, no tool dispatch, no hooks. The atomic counterpart to the operator-facing `ChatRequest` Display action.

Use it for: alarm probable-cause narration, message translation, text rephrase, classification, summarization, and any other "run this through an LLM and give me the answer" task that does not need follow-up turns.

Signatures

```
// Synchronous — returns the full reply envelope JSON.
string TK.AIExecute(string query);

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

Parameter	Type	Notes
`query`	string	Either a plain text question, or a structured query JSON with `system` / `user` / `context` / `metadata` fields. See **Query format** below.

Returns	Notes
string (JSON)	The reply envelope. Always well-formed JSON, parseable with `JObject.Parse`. See Local AI Reply Envelope Schema.

Where it can be called from

`TK.AIExecute` is a Toolkit static — callable from any script context. But the resolution and execution happen in whichever process is running the calling code, and meaningful execution requires a server-domain ObjectServer. Practical surface:

Context	Works?	Notes
Server.Class method	Yes	The canonical caller. Class methods invoked from Server-domain trigger contexts (alarm callbacks, dataset cycles, custom server logic) execute in TServer with full server-side state.
Server-trigger Script.Task	Yes	Tasks running under `ScriptTaskServer` have a server-side ObjectServer.
Report generator (server-side report rendering)	Yes	ReportServer hosts a server-side ObjectServer; reports may call `TK.AIExecute` to add narrative summaries.
Alarm-event callback (server-domain)	Yes	Alarm callbacks fire under TServer.
Display CodeBehind (WPF or HTML5 client)	No (returns error envelope)	Display CodeBehind runs at the client. The ambient ObjectServer is the client-side instance; secret resolution short-circuits, the LLM endpoint URL may not be reachable, and the call falls back to a normal HTTP-error envelope. Use the `ChatRequest` Display action to talk to Local AI from a Display.

The platform does NOT add a runtime gate that blocks client-side callers — instead, the existing platform safety nets ensure no silent secret leak. A misrouted client-side call returns `status="error"` with an HTTP-error warning, never a successful POST with degraded credentials.

Query format

Two accepted shapes — the simple form and the structured form.

Simple form (plain text)

Pass any plain string. The platform wraps it as the user message of a single-turn LLM exchange:

```
string reply = TK.AIExecute("Translate to French: Pump 1 is offline.");
```

Structured form (JSON object)

Pass a JSON object with any combination of these top-level fields:

Field	Required?	Role
`user`	Required when using structured form	The user-role message — the actual question or instruction.
`system`	Optional	The system-role message — sets the LLM's persona, voice, output constraints.
`context`	Optional	Arbitrary structured data the LLM should consider when answering. Typically a JSON object holding tag values, sensor readings, or domain context. Serialized into the LLM prompt.
`metadata`	Optional	Reserved for caller-side annotation; the platform does not currently consume it.

```
var query = new JObject
{
    ["system"]  = "You are a SCADA alarm-message translator. Preserve tag names and " +
                  "numeric values verbatim. Keep replies operator-friendly.",
    ["user"]    = "Translate to French: Pump 1 is offline.",
    ["context"] = new JObject { ["targetLanguage"] = "fr" }
};

string reply = TK.AIExecute(query.ToString());
```

Reply envelope

Every call returns the same JSON shape. Field reference and status semantics: Local AI Reply Envelope Schema.

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": [],
  "latencyMs": 832,
  "warnings":  []
}
```

Note: `toolTrace` is always an empty array (`[]`) for atomic calls — the atomic path has no tool surface. This differs from the `ChatRequest` action, which may dispatch tools and populate `toolTrace` with each one.

Exception model — never throws

`TK.AIExecute` never throws. Every failure path — invalid context, model offline, network error, malformed query, gate disabled, internal exception — returns a well-formed reply envelope with `status` set to `error` or `disabled` and an explanatory entry in `warnings[]`.

Failure modes:

Situation	`status`	`warnings` entry
Invalid query JSON	`error`	`"Invalid query JSON: <parser message>"`
Structured query missing `user` field	`error`	`"Query missing required field 'user'."`
`SolutionSettings.ModelEnabled = false`	`disabled`	`"Local AI master kill-switch (ModelEnabled) is off."`
LLM endpoint URL empty or unresolvable	`error`	`"AI endpoint URL is empty (LocalAIDefaults misconfigured?)."`

Wall-clock budget (60s) exceeded	`truncated`	`"LLM POST wall-clock budget (60s) exceeded."`
HTTP non-2xx response	`error`	`"LLM endpoint HTTP error: <status code> <reason>"`
Network failure (connection refused, DNS, etc.)	`error`	`"LLM endpoint HTTP error: <exception message>"`
Other unhandled exception	`error`	`"<exception type>: <message>"`

In all cases the envelope is well-formed and parseable. Customer scripts can rely on `JObject.Parse(reply)` never throwing on a Local AI reply.

Master gates

The atomic path applies only ONE master gate:

1. `SolutionSettings.ModelEnabled = true` — kill-switch must be ON. Off returns `status="disabled"` immediately, with `latencyMs = 0`.

Atomic does NOT consult `ModelOptions` — the bitmask gates the tool surface, and the atomic path has no tools. A solution with `ModelEnabled = true` and `ModelOptions = 0` (every bit off) can still call `TK.AIExecute("summarize this")` and receive a normal reply.

This asymmetry is deliberate: it lets a solution reserve the tool-equipped chat surface for selected operator panels (master tool bit OFF) while still using `TK.AIExecute` for pure-language tasks like translation and summarization.

Worked examples

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

When a critical alarm fires, an operator typically scans several related tags to form a hypothesis about the actual cause. This Server.Class collects those tags automatically 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 the eight numeric inputs against background domain knowledge of pump failure modes and selects the explanation that fits this specific snapshot.

Example 2 — Multi-language alarm message translation

Critical alarm message authored in English; site operators read other languages. Translate while preserving technical terms 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;
}
```

Example 3 — Operator note rephrase

Free-text operator notes get filed verbatim, then asynchronously rephrased into a consistent house-style summary for the shift report.

```
public void RephraseOperatorNote()
{
    string raw = @Tag.OperatorNote.LastEntry;
    if (string.IsNullOrWhiteSpace(raw))
    {
        @Tag.OperatorNote.LastEntryRephrased = "";
        return;
    }

    var query = new JObject
    {
        ["system"] = "Rephrase the operator's free-text note into ONE concise sentence in " +
                     "third-person past tense. Preserve every numeric value and tag reference. " +
                     "No editorial commentary.",
        ["user"]   = raw
    };

    string reply = TK.AIExecute(query.ToString());
    string text  = JObject.Parse(reply).Value<string>("text") ?? raw;

    @Tag.OperatorNote.LastEntryRephrased = text;
}
```

Synchronous vs async

`TK.AIExecute` is synchronous and wraps the async overload via the platform helper `T.Library.tRPC.AsyncHelpers.RunSync`. Use the synchronous form from a Server.Class method that needs a result inline; use `TK.AIExecuteAsync` from native async script contexts.

Do NOT use raw `.Result` or `.GetAwaiter().GetResult()` on `TK.AIExecuteAsync` — those patterns deadlock under a UI SynchronizationContext or surface `AggregateException` instead of unwrapping the inner exception. The synchronous overload handles both correctly via the platform helper.

What TK.AIExecute does NOT do

- Does not stream replies. Each call returns one complete envelope when the model finishes.
- Does not maintain a transcript. Every call is independent — the LLM sees only the supplied query.
- Does not dispatch tools. The LLM cannot read tags, browse the namespace, or query alarms during a `TK.AIExecute` call. For tool-equipped chat, use the `ChatRequest` Display action.
- Does not raise `OnBeforeChat` or `OnAfterChatReply` hooks. Hooks are scoped to the chat surface only.
- Does not retry on transient failure. A failed call returns the error envelope immediately; the calling code decides whether to retry.
- Does not throw on failure (see **Exception model**).

See also

- Local AI — the parent reference, includes a quick-start.
- ChatRequest Action Reference — the operator-chat counterpart.
- Local AI Configuration — endpoint, master enable, and bitmask configuration.
- Local AI Reply Envelope Schema — full reply schema reference.

In this section...