AI Runtime Connector

Enable AI-powered industrial automation through Model Context Protocol integration.

- Name: MCPServer
- Version: 1.0.0.0
- Interface: TCP/IP
- Configuration:
  - Scripts / Classes

Overview
Prerequisites
Configuration
Local Access: Same Machine
Remote Access: HTTP
Applying Changes
Available Tools
Custom Methods
Deprecated / Removed Tools
Best Practices
Troubleshooting
Related Documentation
Change Log

Overview

The AI Runtime service bridges FrameworX solutions with AI language models, enabling intelligent automation assistance while maintaining industrial-grade safety. This connector exposes your solution's live data and functionality as structured tools that AI models can invoke.

> ***Note:*** *This connector is for querying live data from running solutions (connects to TServer.exe). For AI-assisted solution configuration, see* ***AI Designer Connector****.*

Key Capabilities

- **Real-time Data Access** — Query live tag values, historian data, and alarm states through AI
- **Unified Namespace Browsing** — Explore your tag structure and discover available data
- **Historical Analysis** — Query historian for trend analysis and past behavior investigation
- **Alarm Monitoring** — Check active alarms and query alarm history
- **Custom Methods** — Extend with your own solution-specific AI tools
- **Platform Agnostic** — Works with Claude, GitHub Copilot, and other MCP-compatible AI models

---

Integration Architecture

AI Model		MCP Protocol		FrameworX Solution
Claude, GPT, GitHub Copilot		Model Context Protocol		TServer.exe (Runtime)

Structured Methods

Tag Data Access	Historian Queries	Alarm Monitoring	Custom Methods
Live values from UNS	Historical time-series data	Active & historical alarms	Your solution-specific tools

This diagram shows an AI model (such as Claude or GPT) communicating with a FrameworX solution through the MCP Protocol, which acts as a standardized bridge for exchanging context and invoking tools. FrameworX then exposes structured methods that the model can call to access tag data, query historical records, monitor alarms, and run custom operations.

When to Use AI Runtime

Use Case	Example
Process monitoring	"What is the current temperature in Tank1?"
Alarm investigation	"Are there any active alarms? What's causing them?"
Historical analysis	"Show me the temperature trend for the last 24 hours"
Data discovery	"What tags are available under Plant1/Line2?"
Operational questions	"What is the system uptime?"

Why Custom Tools Matter

Some platforms may offer only built-in MCP tools that let AI read or write tags, or run queries, or perform specific tasks in the platform. In practice, this approach has limited value:

- Real solutions have hundreds of equipment items and thousands of tags
- Operators don't memorize tag names and their needs goes beyond plain data, it requires context, aggregation, and analysis
- Open access to running systems creates safety concerns and privacy issues
- LLM hallucination risk makes direct control inappropriate

FrameworX approach: Custom Runtime Tools

Because FrameworX compiles .NET code at runtime, you can create domain-specific tools that expose exactly what you want, with descriptions that guide AI to correct usage. External applications query at the semantic level: "get tank levels" instead of "read Tag.Area1.Tank3.LT001.PV".

This gated access model means:

Generic Tools	Custom Tools
AI needs to know exact tag paths	AI uses semantic descriptions
Full system access (security risk)	You control what's exposed
Requires tag naming knowledge	Natural language queries
Hallucination can cause errors	Validation in your code

Prerequisites

- FrameworX 10.1 or later
- Cross-platform .NET Desktop Runtime matching your FrameworX version: .NET 8 for 10.1.4 and earlier, .NET 10 for 10.1.5 and later
- Claude Desktop or compatible MCP client (In this documentation it might mention "AI Client", but all the tests were done using Claude from Anthropic)
- Running FrameworX solution (TServer.exe)
- Network connectivity

Configuration

The MCP Server starts automatically when your solution runs. To connect an AI client you install the FrameworX extension that matches how the client reaches the runtime. No manual editing of the AI client configuration file is required.

The FrameworX installer creates the setup folder on the machine where FrameworX is installed:

`C:\Users\<user>\Documents\FrameworX\AISetup`

This folder holds one ready-to-install extension bundle per connection method.

Method	Use Case	Extension to install
Local	AI client on the same machine as the solution	`frameworx-runtime-mcp.mcpb`
Remote (HTTP)	AI client on a different machine	`frameworx-runtime-mcp-http.mcpb`

Local Access: Same Machine

Use this method when the AI client runs on the same machine as your FrameworX solution.

Step 1 — Install the extension

1. Open Claude Desktop
2. Go to **File Settings Extensions**
3. Click **Advanced Settings**
4. Click **Install Extension**
5. Select `frameworx-runtime-mcp.mcpb` from the `AISetup` folder
6. Click **Install**, then **Install** again on the confirmation popup

> Anthropic updates the Claude Desktop settings interface regularly. If these menu entries are not where described, check Claude's official documentation for the current location.

Step 2 — Adjust the installation folder or port (only if not default)

The extension points at the default FrameworX installation folder, `C:\Program Files\Tatsoft\FrameworX`, and at the default runtime port `3101`. If either differs on your machine, change them here:

1. Go to **File Settings Extensions**
2. Find **FrameworX AI Runtime** in the list and click **Configure**
3. Set the **FrameworX installation folder** to your install path
4. Set the **port** to the TServer port your solution is running on

> The port configured in the extension must match the port your FrameworX runtime is actually listening on. FrameworX uses `3101` by default. If you change the port your solution runs on, change it in the extension's **Configure** panel as well, then restart Claude Desktop.

Step 3 — Restart Claude Desktop

Fully quit and reopen Claude Desktop so the extension is loaded. See **Applying Changes** below.

Remote Access: HTTP

Use this method when the AI client runs on a **different machine** from the FrameworX solution. The steps are the same as the local method, with one addition: an HTTP bridge must be running on the FrameworX machine.

Step 1 — Install the extension

On the AI client machine, follow the same procedure as the local method, selecting `frameworx-runtime-mcp-http.mcpb` instead.

> **Prerequisite:** Node.js (LTS) must be installed on the AI client machine. The HTTP bundle runs the `mcp-remote` bridge, which is an npm package. The local (stdio) bundle does not need it.

1. Open Claude Desktop
2. Go to **File Settings Extensions**
3. Click **Advanced Settings**
4. Click **Install Extension**
5. Select `frameworx-runtime-mcp-http.mcpb` from the `AISetup` folder
6. Click **Install**, then **Install** again on the confirmation popup

Step 2 — Start the HTTP bridge on the FrameworX machine

On the machine where FrameworX is installed, run `RuntimeMCPHttp.exe` from the installation folder:

```
C:\Program Files\Tatsoft\FrameworX\fx-10\net10.0\RuntimeMCPHttp.exe
```

Adjust the runtime folder to match your FrameworX version (`net8.0` for 10.1.4 and earlier, `net10.0` for 10.1.5 and later). If you installed FrameworX somewhere other than the default location, use that path instead.

Leave the console window open. From FrameworX 10.1.6, the bridge connects only to a runtime you name. To name it at startup, set `Runtime.Host` and `Runtime.Port` in `RuntimeMCPHttp.json` (see **Advanced: HTTP Bridge Settings** below). The bridge then connects as soon as the runtime is running, and the start order of the bridge and the solution does not matter. With no runtime named in the file, ask the AI to connect to a solution and execution profile. The AI calls `runtime_connect` with both values and binds the session to the named runtime. Until then, the runtime tools answer with a no-target message and instructions to connect.

In 10.1.5, the bridge starts looking for a FrameworX runtime immediately. As soon as a solution is running, it connects automatically and the AI client has a live remote connection to the runtime. Order does not matter: you can start the bridge before or after the solution.

Step 3 — Adjust the installation folder or port (only if not default)

Go to **File Settings Extensions**, find **FrameworX AI Runtime (HTTP)** and click **Configure**. This variant takes two values: **RuntimeMCPHttp Host**, the address of the machine running the bridge (default `127.0.0.1`), and **RuntimeMCPHttp Port**, the bridge port (default `10120`). Note that this is the bridge port, not the TServer port: the TServer host, port, and credentials are configured in `RuntimeMCPHttp.json` on the runtime machine.

From FrameworX 10.1.6, enter the bridge's `X_API_KEY` in the extension's **API Key** setting to connect from another computer.

Step 4 — Restart Claude Desktop

Fully quit and reopen Claude Desktop so the extension is loaded.

Advanced: HTTP Bridge Settings

From FrameworX 10.1.6, the defaults accept connections from the local machine only. In 10.1.5, the defaults work for a trusted local network. To change the listen port, set an API key, or enable TLS, edit this file on the FrameworX machine **before** starting the bridge:

```
C:\Users\Public\Documents\FrameworX\MachineSettings\RuntimeMCPHttp.json
```

Setting	Description

`X_API_KEY`	Optional authentication password. From FrameworX 10.1.6, the bridge accepts connections from other machines only when this key is set.
`CertFileName`	SSL certificate file (for HTTPS)
`CertPass`	SSL certificate password
`CertHash`	SSL certificate thumbprint (alternative to file)
`ListenPort`	HTTP endpoint port (default: 10120)
`Runtime.Host`	FrameworX runtime host (default: localhost). From FrameworX 10.1.6, the bridge connects at startup only when both `Runtime.Host` and `Runtime.Port` are set. Otherwise it waits for `runtime_connect`.
`Runtime.Port`	FrameworX runtime port (default: 3101). From FrameworX 10.1.6, set it together with `Runtime.Host`.
`Runtime.Username`	Runtime authentication user
`Runtime.Password`	Runtime authentication password

Security note: from FrameworX 10.1.6, the bridge listens on the local machine only while `X_API_KEY` is empty. When you set `X_API_KEY` and leave `ListenLoopbackOnly` at `false`, the bridge listens on all network interfaces and requires the key on every request, so choose a strong key. In 10.1.5, set a strong `X_API_KEY` if the bridge is reachable over a network. Leave `CertFileName` empty to use plain HTTP.

Multiple Connections: to serve multiple runtimes from one machine, start the bridge with the `/instance:<number>` argument and add matching numbered settings (for example `X_API_KEY2`, `ListenPort2`, `Runtime2`).

Applying Changes

Claude Desktop loads extensions once, when the process starts. After installing an extension, reinstalling an updated one, or changing values in the **Configure** panel, you must fully restart the client. Opening a new chat is not enough. The restart of the AI Client is necessary after running the solution as well.

- **Windows:** Close the process in the Task Manager. Only closing the AI Client window is not enough.

Verifying Connection

1. In the chat, click in the '+' button > Connectors > See if "FrameworX AI Runtime" is listed there.
2. Ask the AI a simple question like "What is the server uptime?"

Available Tools

AI Runtime provides built-in platform tools plus solution-authored custom tools, organized by category. The tool count your AI client reports is the sum of both. Each category is gated independently in **Solution Settings Data Servers MCP for Runtime**; the Info tools (`runtime_connect`, `runtime_describe_binding`) are gated only by the master toggle so AI clients can bootstrap and verify the target before any sub-category is enabled.

Info (2 tools)

Tool	Purpose
runtime_connect	Connect to a FrameworX runtime and receive the Context Package (solution identity, profile, namespaces, quality codes, tool workflows, objType schemas, custom tools). **Call this first** at the start of every session. No-args returns the current Context Package for the runtime this MCP is bound to. On retargetable instances, stdio or HTTP, accepts optional `solution` and `profile` arguments to rebind to a different running TServer before returning.
runtime_describe_binding	Pre-auth introspection — calls the anonymous `/health` endpoint and returns the runtime's binding identity: solution name, execution profile (on supporting builds), version, uptime, and start time. Requires **no authentication**. Use it *before* authenticated calls to confirm the target runtime is bound to the solution you expect. A no-argument call probes the runtime this MCP is bound to; pass `tserver_url` to probe an arbitrary running runtime without rebinding (useful for multi-runtime verification). From FrameworX 10.1.6, every response, success or error, carries an `mcp_host` block describing the MCP server which answered: `transport` (`stdio` or `http`), `listen_port`, `listen_scope` (`stdio`, `loopback` or `any_interface`), `local_only` and `instance`. Use it to tell which of several connector instances you reached, and to check `local_only` before calling the Client Session tools. `transport` names the MCP transport, not the URL scheme.

UNS Tools (5 tools, gate: Enable UNS Tools)

Tools for querying live data and structure from the runtime Object Model — the hierarchy that includes the Unified Namespace (`Tag.*`) plus the built-in namespaces (Server, Client, Alarm, Device, Historian, Dataset, Script, Display, Layout, Security, Report, Info, Enumeration, TagProvider).

Tool	Purpose
runtime_get_value	Get the current LIVE value of a tag or runtime object, including quality (0=Bad, 64=Uncertain, 192=Good), timestamp, and metadata (description, units, min/max). For historical values, use `runtime_get_tag_history`.
runtime_browse_object_model	Browse the live runtime object-model tree — returns child nodes at the specified path. Works across all roots: `Tag` entries at the Tag root, full tag names such as `Plant1/Line2/Motor1` or folders; `Tag.Plant1/Line2` tags inside a folder; `Tag.Pump1` UserType members; `Server`, `Alarm.Group`, `Device.Channel`, and so on. Bounded by `max_results`.
runtime_search_uns	Ranked semantic search across the live UNS — UserTypes, Tags, members, Enumerations, and AssetTree folders. Scores across Name, DisplayText, Labels, SourceIri, and Attributes annotation content. Returns ranked matches with `matchReason`, `snippet`, and `score`. Tag matches include live value, quality, and timestamp.
runtime_describe_object	Describe a single UNS object — inheritance chain, derived UDTs, and (for Tag paths) live current value, quality, timestamp, and alarm state. Pass either `path` or `iri`. The `iri` form reverse-looks up by external SourceIri from an ontology import.
runtime_list_instances	List all live tags of a given UDT (UserType), with current value, quality, and timestamp for each. Optional `include_derived=true` expands to subclass UDTs. Optional `name_mask` narrows to a wildcard pattern (e.g., `Motor*`, `Area1/*`, `*_Pump`).

Alarm Tools (2 tools, gate: Enable Alarm Tools)

Dedicated alarm-query tools that hit the alarm subsystem's state machine and historian. Reading an `Alarm.*` namespace value through `runtime_get_value`, or reading a tag's current alarm-state through `runtime_describe_object`, stays under *UNS Tools* — those are UNS data reads, not dedicated alarm queries.

Tool	Purpose
runtime_get_active_alarms	Get the active-alarm list: alarms in alarm now (State 1 or 3) and alarms back to normal and still waiting for acknowledgement (State 2). Returns tag name, message, priority, active time, area, and group. For past alarm events, use `runtime_query_alarm_history`.
runtime_query_alarm_history	Query historical alarm records from the Alarm Historian database. This is a typed request with no SQL: pass `startDate` and `finishDate`, plus the optional filters `tag_name`, `area` and `priority`. The server resolves the alarm history table and the database dialect. Rows come back newest first, up to `max_rows` (default 100, maximum 1000).

Historian Tools (2 tools, gate: Enable Historian Tools)

Tools for querying historical time-series data.

Tool	Purpose
runtime_get_tag_history	Query historical time-series data for a tag from the Historian database. The tag must be configured for Historian storage. Returns raw stored samples (truncated at 2000 points). For the current live value, use `runtime_get_value`; for summarized or down-sampled data, use `runtime_get_aggregated_history`.
runtime_get_aggregated_history	Bucketed aggregation over historical time-series data — the kernel reduces each time bucket into a single value rather than returning raw samples. Specify an ISO-8601 time range and `interval` bucket width (e.g., `PT5M`, `PT1H`, `P1D`). Seven aggregation kinds: **Average** (time-weighted, OPC-UA Part 13), **Minimum**, **Maximum**, **Total** (time-weighted integral), **Count**, **StandardDeviation** (population), and **Range**. Samples below quality 64 (Bad) are excluded; empty buckets return null. Use it when raw-sample counts would exceed the 2000-point cap, when summary statistics are needed, or for evenly-spaced chart points.

Wait Tool (1 tool)

From FrameworX 10.1.6. Read-only. Available in the stdio and HTTP connectors. The `tag` condition needs **Enable UNS Tools**, the `alarm` condition needs **Enable Alarm Tools**, and the `module` condition needs the master toggle only.

Tool	Purpose

runtime_wait_for	Block until a live runtime condition is true, then return the triggering event, or return a structured `TIMEOUT`. Use it in automated tests instead of polling values or screenshots. Parameters: • `kind`: `tag`, `alarm` or `module`. • `target`: a tag path such as `Tag.Line1/Tank1/Level`. For `alarm`, the alarm source tag (all alarm items on the tag) or one item as `Alarm.Item.<ItemName>`. For `module`, a module name such as `Alarm` or `Historian`. • `comparison`: for `tag`, `eq`, `ne`, `gt`, `ge`, `lt`, `le` or `changed`. For `alarm`, `active` or `normalized`. For `module`, `paused` or `running`. • `value`: the operand of the tag comparisons. • `timeout_seconds`: default 30, maximum 300. • `allow_already_normal`: for `alarm` with `normalized` only. With the default `false`, the wait first sees the alarm in alarm and then back to normal, so it never passes on an alarm which never fired. `true` accepts an alarm already normal when the wait starts. A success returns `result: SATISFIED` with `observedValue`, `elapsedMs` and `polls`. A timeout returns `result: TIMEOUT` with `lastObservedValue`. The alarm condition reads the alarm's own state, not the active alarm list. It completes on an alarm back to normal and still unacknowledged, and it sees alarms whose group keeps them off the active list. Alarm results add `alarmSource`, `alarmStatePath`, `alarmStateRaw`, `sawActive` and `allowAlreadyNormal`. On a timeout, `sawActive` tells whether the alarm fired at all.

Client Session Tools (Development only)

Tools for inspecting and driving connected client sessions during local development and automated testing. They are not general operations tools and are not exposed through the **MCP for Runtime** gate panel. From FrameworX 10.1.6a, the AI Runtime server lists them only when it starts with `/AllowTestTools:true`, which is off by default. On the HTTP server (`RuntimeMCPHttp`), the option takes effect only when the server accepts local connections only. Each one answers only when *all* of the following hold: the runtime's `QaTestMode` gate is active, the caller reaches the runtime over true loopback (`127.0.0.1` / `::1`), the target client's connection and the address it reports are both true loopback, and this MCP host is on a transport a remote caller cannot reach (STDIO always; HTTP only when launched with `ListenLoopbackOnly=true`). When a gate refuses, the tool returns an `errorCode` and a message instead of data.

Tool	Purpose
runtime_list_client_sessions	List the client sessions currently connected to the running TServer — one row per connected client (Rich Client, Smart Client, HTML5, Designer, Data Access, protocol drivers). Read-only flat projection: session GUID, client type, additional name, whether it is a view client, whether it is connected over loopback, logged-in user, UTC connect time, and client version. Call it first — the other tools address a session by the GUID it returns.
runtime_client_screenshot	Capture a screenshot of ONE connected WPF Rich Client session's window, addressed by its session GUID. Pass an empty display name to capture the whole open layout, or a display name to capture that single display. Returns a local file path on the runtime host. **WPF Rich Client only** — the HTML5 WebClient cannot be captured and returns `SCREENSHOT_TIMEOUT`.
runtime_client_input	Inject an operator-input value into ONE connected client session, addressed by its session GUID. The value is delivered into that client's own commit routine, the one that runs when an operator types into the named input control and tabs away. It does **not** write the tag directly. From FrameworX 10.1.6, on a WPF client (Rich Client or Smart Client) the tool reports what the client did: `committed` (value accepted), `unchanged` (the control already held the value), or a refusal code such as `ELEMENT_NOT_FOUND`, `CONTROL_NOT_INPUT` (not an input control, for example a button) or `INPUT_REJECTED` (out of range, permission, or a cancelled confirmation). Use `runtime_client_invoke` to click a button. On the HTML5 client the input runs without a report (outcome `unreported`), so verify the result by reading the tag back with `runtime_get_value`. In 10.1.5, a successful call means the injection was *enqueued*, not committed: the client can still reject it. Verify the commit by reading the tag back with `runtime_get_value`. From FrameworX 10.1.6a, the tool acts on a client while electronic signature is off for the client's logged-on user, so it never answers a signature prompt. On a WPF client with electronic signature on, it returns the outcome `ESIGN_ENABLED`.
runtime_client_invoke	From FrameworX 10.1.6. Click ONE element on a connected WPF client (Rich Client or Smart Client), addressed by its session GUID, without the physical mouse. The element's own click actions run, with the same security check and confirmation prompt a human click raises. A disabled element returns `CONTROL_NOT_ENABLED`. When a confirmation prompt opens, the call returns `INVOKE_OUTCOME_TIMEOUT`, and the action runs only if a person answers the prompt on the client. The HTML5 client is not supported in this version and returns `CLIENT_TYPE_NOT_SUPPORTED`. From FrameworX 10.1.6a, the tool acts on a client while electronic signature is off for the client's logged-on user, so it never answers a signature prompt. With electronic signature on, it returns the outcome `ESIGN_ENABLED`.

A screenshot can expose any on-screen content, and an input injection or a click drives a live client. All of these tools are restricted to localhost Development for that reason. None of them is reachable from a remote AI client over the HTTP bridge unless that bridge was deliberately launched loopback-only.

Custom Tools (gate: Enable Custom Tools)

Solution-authored `[McpServerTool]` methods on Scripts ? Classes of type **MCP Tool**. See the Custom Methods section below for authoring guidance.

Reading Live Values (runtime_get_value):

- "What is the current value of Tag.Temperature1?"

- "Get the value of Tag.Plant1/Line2/Pressure"
- "What is the Server.SystemMonitor.CPUUsage?"
- "How many active alarms are there?" (uses Alarm.TotalCount)

Browsing the Object Model (runtime_browse_object_model):

- "What tags are available under Tag.Plant1?"
- "Show me the members of Tag.Pump1" (for UserType tags)
- "What alarm groups exist?"
- "Browse the Server namespace"

Searching the UNS (runtime_search_uns):

- "Find all tags containing 'Temperature'"
- "Search for UserTypes related to 'motor'"
- "Find anything with 'pump' in the description or labels"

Historical Analysis (runtime_get_tag_history):

- "Show me the temperature history for today"
- "Get the historian data for Tag.Pressure from 8am to noon"
- "What was the highest temperature value today and when did it occur?"

Active Alarms (runtime_get_active_alarms):

- "Are there any active alarms?"
- "What alarms are currently in alarm state?"
- "Show me all unacknowledged alarms"

Alarm History (runtime_query_alarm_history):

- "Show me alarms from the last 24 hours"
- "How many alarms were acknowledged today?"
- "Query the alarm history for Tag.Temperature1"
- "What alarms occurred during the night shift?"

Custom Methods

You can extend the MCP server with your own solution-specific tools by creating custom methods in Scripts.

> ***Tool names in runtime:*** *When naming your runtime tools, it's strongly recommended that you keep the pattern* `runtime_<toolName>`*. When AI is also running AI Designer, or other third-party tools, that helps to correctly identify the right tool to use. For adherence with MCP standards use snake_case for method names (e.g.,* `runtime_get_tank_level`*), instead of RuntimeGetTankLevel that would be a typical C# convention.*

Creating Custom Methods

1. Navigate to **Scripts Classes**
2. Create a new class with type **MCP Tool**
3. Define methods using MCP decorators:

```
[McpServerTool, Description("Get current tank level for a specific tank")]
public string runtime_get_tank_level(
    [Description("Tank identifier (1-4)")] string tank_id = "1")
{
    return @Tag[$"Tank{tank_id}_Level"].ToString();
}
```

Method Structure:

Element	Purpose
`[McpServerTool]`	Marks method as MCP tool
`[Description("...")]`	Explains what the tool does (shown to AI)
Method name	Tool name (use snake_case)
Parameter descriptions	Explains expected input to AI
Return value	Result sent to AI

Custom Method Examples

Simple Value Retrieval:

```
[McpServerTool, Description("Get production count for today")]
public string runtime_get_production_count()
{
    return @Tag.Production.TodayCount.ToString();
}
```

With Error Handling:

```
[McpServerTool, Description("Get data for a specific tag with validation")]
public string runtime_get_tag_data(
    [Description("Full tag path")] string tag_name)
{
    try
    {
        if ([email protected](tag_name))
            return $"Error: Tag '{tag_name}' not found";

        return @Tag[tag_name].ToString();
    }
    catch (Exception ex)
    {
        @Info.Trace($"MCP Error: {ex.Message}");
        return "Error: Unable to retrieve data";
    }
}
```

Aggregated Data:

```
[McpServerTool, Description("Get summary of all tank levels")]
public string runtime_get_all_tank_levels()
{
    var result = new StringBuilder();
    for (int i = 1; i <= 4; i++)
    {
        var level = @Tag[$"Tank{i}_Level"].Value;
        result.AppendLine($"Tank {i}: {level}%");
    }
    return result.ToString();
}
```

> ***Important:*** *After creating or modifying custom methods, fully restart the AI client (close from Task Manager, not just the window).*

Deprecated / Removed Tools

Customers upgrading from 10.1.4 should update any AI prompts, solution-authored wrappers, or integration scripts that reference the old tool names. Bit positions in `ModelOptions` are unchanged — existing solutions decode identically; only the tool surface and category labels changed.

Old name (10.1.4 and earlier)	10.1.5 replacement	Notes
`runtime_get_info`	`runtime_connect`	Same payload on no-args. HTTP retargetable instances accept optional `solution` / `profile` to rebind to a different running TServer.
`runtime_set_target`	`runtime_connect (solution, profile)`	Merged as the with-args branch of `runtime_connect`.
`runtime_browse_uns`	`runtime_browse_object_model`	Same parameters plus a bounding `max_results`. Renamed because the tool walks the Object Model roots, not just the strict UNS (`Tag.*`).
`runtime_get_object_context`	`runtime_describe_object`	Same single-object intent; absorbs external-IRI lookup via a new `iri` parameter.
`runtime_find_by_iri`	`runtime_describe_object(iri=...)`	Merged into `runtime_describe_object` — different lookup key, same single-object response.
`runtime_list_by_type`	`runtime_list_instances`	Renamed. Adds `max_results` and `name_mask` wildcard filter.
`runtime_list_derived_types`	— (removed)	Redundant: `runtime_describe_object` already returns `derivedTypes` in its context block.
`runtime_search_tags`	`runtime_search_uns`	Broader scope — searches UserTypes, members, Enumerations, and AssetTree folders in addition to Tags.

Best Practices

For AI Queries

- Be specific about tag paths when asking for values
- Use "active alarms" for current state, "alarm history" for past events
- Specify time ranges clearly for historical queries
- Ask the AI to browse the namespace if you're unsure of tag names

For Custom Methods

- Use descriptive names and descriptions — the AI relies on these
- Include parameter validation and error handling
- Return clear error messages that help the AI understand what went wrong
- Follow snake_case naming convention for consistency
- Keep methods focused on single tasks

Security Considerations

- Use `X_API_KEY` authentication for HTTP connections
- Configure appropriate runtime user permissions
- Be cautious with write operations — consider read-only access for AI
- Review custom methods for security implications
- From FrameworX 10.1.6a, the AI Runtime server offers `runtime_force_switchover`, which commands a redundancy switchover, only when it starts with `/AllowControlActions:true`, which is off by default. On the HTTP server, the option takes effect only when the server accepts local connections only. The tool is marked as destructive, so AI clients that confirm destructive actions ask before it runs.

Troubleshooting

AI Runtime server not starting

- Verify the cross-platform .NET Desktop Runtime matching your FrameworX version is installed (.NET 8 for 10.1.4 and earlier, .NET 10 for 10.1.5 and later)
- Check firewall settings for the configured port
- Confirm the FrameworX installation folder is correct in the extension's **Configure** panel

AI cannot access methods

- Ensure solution is running (the TServer.exe application is running)
- Verify AI client configuration matches server settings
- Restart AI client completely (close via Task Manager)
- Check that MCP decorators are properly applied to custom methods

Connector process stopped or kept running unexpectedly

- From FrameworX 10.1.6, each stdio AI Runtime connector process writes a small lifetime log, `McpLifecycle_<Connector>_<pid>.log`, in the product TraceLogs folder.
- The log records which launching process and grandparent the connector watches, a fallback to watching the launching process only, a watch it failed to start, and why it shut down after a watched process ended. A clean disconnect by the AI client records no exit reason.
- Each file stops growing at 64 KB, and files older than 30 days are removed.
- The HTTP bridge (`RuntimeMCPHttp`) does not watch a launching process.

Data not updating

- Confirm tags are properly configured in the solution
- Verify real-time database connectivity
- Check custom method error handling for exceptions

Custom methods not appearing

- Verify class type is set to "MCP Tool"
- Check for syntax errors in method code
- Restart AI client completely after changes

HTTP connection failing

- Confirm `RuntimeMCPHttp.exe` is running on the FrameworX machine
- Verify ListenPort is not blocked by firewall
- Check X_API_KEY matches between server and client
- For HTTPS, verify certificate is properly configured
- Test with `http://localhost:10120` first before remote access

- From FrameworX 10.1.6, set `X_API_KEY` on the bridge for access from another machine. With no key, the bridge accepts local connections only.

Related Documentation

Quick Start Tutorial

- **AI Runtime Tutorial** — Creating and deploying MCP Tools

Example Implementation

- **SolarPanels MCP Demo** — Full solution demonstrating MCP Tools with solar panel monitoring

Technology Information

- **AI-Ready by Design** — Platform architecture for AI integration

Reference Information

- **Scripts Module Reference** — Complete scripting documentation

AI Designer

For AI-assisted solution configuration (creating tags, displays, alarms):

- **AI Designer Connector** — Enable AI to configure solutions
- **AI Designer In Action** — See AI Designer capabilities in action

---

Change Log

Version	Date	Changes
1.6a	2026-10	FrameworX 10.1.6a. The Client Session tools are listed when the AI Runtime server starts with `/AllowTestTools:true`. `runtime_force_switchover` is listed when the server starts with `/AllowControlActions:true`, and is marked as destructive. On the HTTP server, both options take effect when the server accepts local connections only. `runtime_client_input` and `runtime_client_invoke` act on a client while electronic signature is off for its logged-on user. The target check reads both the client's connection and the address it reports.
1.6	2026-10	FrameworX 10.1.6. The HTTP bridge and the AI Runtime tools connect only to a named runtime. Adds `runtime_client_invoke` (WPF clients). `runtime_client_input` reports the outcome on WPF clients. The HTTP bridge accepts connections from other machines only when `X_API_KEY` is set. The platform tool count is no longer listed. Adds `runtime_wait_for`. `runtime_describe_binding` returns an `mcp_host` block. The stdio connector writes a lifetime log in TraceLogs.
1.5	2026-07	14 platform tools (was 11). Adds Client Session tools — `runtime_list_client_sessions`, `runtime_client_screenshot`, `runtime_client_input` — for local development and automated testing, gated by `QaTestMode`, true loopback, and a local-only MCP transport.
1.4	2026-07	Extension-based configuration. Local and remote connections are now installed from the ready-made `.mcpb` bundles in `Documents\FrameworX\AISetup`. Manual editing of the AI client configuration file is no longer required; the HTTP bundle carries the `mcp-remote` bridge configuration, which still needs Node.js on the AI client machine.
1.3	2026-06	11 platform tools (was 9)
1.2	2026-05	9 platform tools (was 7) plus solution-authored custom tools. Session bootstrap unified as `runtime_connect` (merges `runtime_get_info` + `runtime_set_target`). Tool renames for scope-accurate categories: `runtime_browse_uns` `runtime_browse_object_model` (walks the Object Model roots); `runtime_search_tags` `runtime_search_uns` (searches UserTypes, members, enums, folders); `runtime_get_object_context` `runtime_describe_object` (absorbs `runtime_find_by_iri` via `iri` parameter); `runtime_list_by_type` `runtime_list_instances` with wildcard `name_mask`. `runtime_list_derived_types` removed (folded into `runtime_describe_object.derivedTypes`). Gate-bit categories relabeled to Enable *Tools.
1.1	2026-02	7 built-in tools; Updated architecture diagram; Added runtime_ prefix best practices.

In this section...