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

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
- Running FrameworX solution (TServer.exe)
- Network connectivity

Configuration

The MCP Server starts automatically when your solution runs. You need to configure your AI client to have the solution **connector**.

Connection Methods

Method	Use Case	Requirements
stdio	AI client on same machine as solution	Local installation
HTTP	AI client on different machine	Network access, optional SSL

Local Access: stdio Configuration

Use `stdio` when the AI client runs on the same machine as your FrameworX solution.

Step 1 — Open the Claude configuration folder

Press **Win + R**, type `%APPDATA%\Claude`, and press Enter.

Step 2 — Edit `claude_desktop_config.json`

Open the file and add the following configuration. Adjust the runtime folder in the path to match your FrameworX version (`net8.0` for 10.1.4 and earlier, `net10.0` for 10.1.5 and later):

```
{
  "mcpServers": {
    "MCPRuntime": {
      "command": "C:\\Program Files\\Tatsoft\\FrameworX\\fx-10\\net8.0\\RuntimeMCP.exe",
      "args": ["/host:127.0.0.1", "/port:3101"],
      "transport": "stdio"
    }
  }
}
```

> The product path may vary depending on where you performed the installation, but this is the default location.
>
> The connector name can be modified changing **MCPRuntime** in the file above to any other name.

Remote Access: HTTP Configuration

Use HTTP mode when the AI client runs on a **different machine** from the FrameworX solution.

Server Setup

1. (Optional) Configure server settings

If you need to change the port, set an API key, or configure TLS, edit this file **before** starting the server:

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

HTTP Settings Reference:

Setting	Description
`X_API_KEY`	Optional authentication password
`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)
`Runtime.Port`	FrameworX runtime port (default: 3101)
`Runtime.Username`	Runtime authentication user
`Runtime.Password`	Runtime authentication password

```
{
  "appSettings": {
    "X_API_KEY": "",
    "CertFileName": "",
    "CertPass": "",
    "CertHash": "",
    "ListenPort": "10120",
    "Runtime": {
      "Host": "localhost",
      "Port": "3101",
      "Username": "guest",
      "Password": ""
    }
  }
}
```

> Security note: Set a strong `X_API_KEY` if the server is accessible over a network. Leave `CertFileName` empty to use plain HTTP.

2. Start the HTTP server

Run "C:\Users\<user>\Documents\FrameworX\Utilities\StartRuntimeMCPHttp.bat" on the machine where your FrameworX solution is running.

Client Setup

Replace `<RemoteIP>` with the IP address or hostname of the server machine.

GitHub Copilot (`.mcp.json`)

```
{
  "servers": {
    "FrameworX Runtime": {
      "type": "http",
      "url": "http://<RemoteIP>:10120",
      "headers": {
        "X-API-KEY": "<your-api-key>"
      }
    }
  }
}
```

Claude Desktop / Claude.ai (`.mcp.json`)

> **Prerequisite:** Node.js must be installed on the client machine.

```
{
  "mcpServers": {
    "FrameworX Runtime": {
      "command": "C:\\WINDOWS\\System32\\cmd.exe",
      "args": [
        "/C",
        "C:\\Program Files\\nodejs\\npx.cmd",
        "-y",
        "mcp-remote",
        "http://<RemoteIP>:10120/mcp",
        "--allow-http"
      ]
    }
  }
}
```

> `--allow-http` is required when not using TLS. For production environments, configure a certificate and use `https://` instead.

Multiple Connections: To connect to multiple runtimes, use `/instance:<number>` argument and matching numbered settings (e.g., `X_API_KEY2`, `ListenPort2`, `Runtime2`).

Verifying Connection

1. Click the LLM connector icon — you should see your solution name
2. Go to **Settings Developer** and verify the connection shows "running"
3. Ask the AI a simple question like "What is the server uptime?"

Available Tools

AI Runtime provides **11 built-in platform tools** plus solution-authored custom tools, organized by category. 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 HTTP retargetable instances, 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).

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

Tools for querying live data and structure from the runtime Object Model — the 12-root hierarchy that includes the Unified Namespace (`Tag.*`) plus the built-in namespaces (Server, Client, Alarm, Device, Historian, Dataset, Script, Display, Security, Report, Info).

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` root tag 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 currently ACTIVE alarms — alarms in alarm state right now, not yet normalized. Returns tag name, message, priority, active time, area, and group. For past normalized alarms, use `runtime_query_alarm_history`.
runtime_query_alarm_history	Query historical alarm records from the Alarm Historian database. Only SELECT statements are allowed; INSERT, UPDATE, DELETE, and other modifying statements are blocked. Available columns: TagName, Message, ActiveTime, AckTime, NormTime, UserName, Priority, Area, GroupName.

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.

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 all 12 runtime 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

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 RuntimeMCP.exe path is correct in AI client config

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

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

- 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

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
1.3	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 all 12 runtime 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...
