Local AI Architecture Reference

Architectural reference for the Local AI capability in FrameworX 10.1.5: the two-path execution model, code organization, master-gate semantics, transcript cache mechanics, tool dispatch loop, and integration points with the rest of the platform.

Version 10.1.5+

Audience and scope

This page is for system integrators, custom-driver authors, OEM partners, and engineers who need to reason about how Local AI is composed inside the FrameworX runtime: which assembly hosts what, which thread runs which work, where gates apply, and which contracts must be honoured when extending or wrapping Local AI.

End users configuring Local AI from the Designer or wiring a chat panel from a Display do not need this page. See Local AI and the user-facing reference children there.

Script API surface naming. The canonical atomic script API is `T.Toolkit.LocalAI.AI.Execute(query)` / `T.Toolkit.LocalAI.AI.ExecuteAsync(query)`. The legacy flat-on-TK aliases `TK.AIExecute` / `TK.AIExecuteAsync` are `[Obsolete]` (CS0618) forwarders that remain callable for backward compatibility. This page refers to the atomic surface as `AI.Execute` throughout; the legacy alias semantics are identical (it forwards to the same underlying methods).

Capability summary

Local AI exposes one platform capability — calling an OpenAI-compatible LLM from inside a FrameworX solution — through two architecturally independent consumer paths:

Path	Surface	Caller context	Stateful?	Tools?	Hooks?
Cached	`ChatRequest` Display action	Operator click ServiceClient TCP Kernel TCPServer dispatch Local AI	Per-Display-panel transcript cache	Yes (UNS / Alarm / Historian / custom)	Yes (`OnBeforeChat` / `OnAfterChatReply`)
Atomic	`AI.Execute` script API	Server-domain script (Server.Class, Script.Task, alarm callback, report generator) Toolkit static	Stateless — every call independent	No (single POST, no tools)	No

Both paths POST to the same OpenAI-compatible endpoint, configured once per solution: the canonical master kill-switch and endpoint blob live on the **Local AI capability row** in `SolutionCapabilities` (the `Enabled` boolean and the `Settings` JSON), while the shared tool-options bitmask stays on `SolutionSettings.ModelOptions`. Both paths return the same reply envelope (see Local AI Reply Envelope Schema).

Two-path model — what is shared and what is divergent

The two paths are deliberately independent code paths sharing a small set of explicit anchors. Code duplication is welcomed where the alternative would be a shared helper that couples them.

Shared by architecture (legitimate coupling)

- **The endpoint configuration** — both paths read `SolutionCapabilities[LocalAI].Settings` JSON via the same defensive accessors on the Local AI service singleton.
- **The master kill-switch** — both paths check `SolutionCapabilities[LocalAI].Enabled` at entry and short-circuit identically when off.
- **The SecuritySecrets resolver** — both paths invoke `Security.CheckAndResolveSecretCommConfiguration` on resolved auth strings; this is a Kernel-side platform service, not a path-specific helper.
- **The reply envelope schema** — both paths produce the same five-field JSON shape (`text`, `status`, `toolTrace`, `latencyMs`, `warnings`).
- **The auth multi-line decoder** — both paths call the platform's pre-existing `ReportWebData.DecodeAuthorizationHeader` static, which is the single source of truth for the `None` / `BearerToken` / `BasicAuth` / `CustomAuth` wire format.

Divergent by design

Concern	Cached path	Atomic path
Implementation class	Sealed orchestrator owning the per-Display-panel transcript	Separate sealed class — no shared methods, no shared state
HTTP-POST helper	Inline in the cached orchestrator	Inline in the atomic class — separate copy, even if today the bodies look similar, so divergence as features grow stays isolated
Master tool-surface bit (`ModelOptions 0x02`)	Required — off `status="disabled"`	Not consulted — atomic has no tools to gate

Per-category tool bits (`0x04` – `0x20`)	AND-ed against master to assemble the tool catalog	Not consulted
Hook chain	Raises `OnBeforeChat` and `OnAfterChatReply`	Does NOT raise hooks — chat-attached behaviour does not apply
Transcript cache	Per-`ClientInfo.Guid`, lazy login-reset, `0x80 EnableChatHistory` sub-gate	None
Tool dispatch loop	Bounded by per-turn cap and 60-second wall-clock budget	Single POST, no loop
Caller wire signature	Receives `(ObjectServer, clientGuid, userName, queryJson)`	Receives `(ObjectServer, queryJson)`

Code organization

Assembly home and the namespace-contribution rule

Local AI is implemented as a sub-namespace inside **`T.Toolkit.dll`** — the `T.Toolkit.LocalAI` namespace, with sources under `FS/Toolkit/LocalAI/`. It is NOT a peer assembly to `T.Toolkit.dll` and it is NOT inside `T.Modules.dll`.

The placement is governed by a platform rule: *code lives in `T.Modules.dll` if and only if it contributes to the runtime Object Model* — Tag namespaces, `eObjType` registration, `ListObj` / `RunObj` entries, user-visible runtime addressability. Local AI has zero Object Model contribution: `AI.Execute` (a static reachable as a Toolkit-sibling entry) and the Display `ChatRequest` action both *consume* the Object Model — they read tags, dispatch `runtime_*` tools — but neither *contributes* a Tag namespace or a runtime root. Therefore Local AI lives as a sub-namespace inside `T.Toolkit.dll` rather than as its own peer assembly or as a runtime Module.

This explains why there is no `AI` namespace in the runtime Object Model in 10.1.5 — Local AI is reachable only as `AI.Execute(...)` from script (with `T.Toolkit.LocalAI` in `NamespaceDeclarations`, or fully qualified) and as the `ChatRequest` Display action. A future `@Solution.AI.*` binding surface is post-10.1.5 work.

Build position and dependencies

Assembly	Local AI sources	Depends on	Consumed by
`T.Toolkit.dll` (`T.Toolkit.LocalAI` sub-namespace)	`FS/Toolkit/LocalAI/*.cs`, included in `T.Toolkit.csproj` via the implicit SDK glob — no explicit `<Compile Include>` entries	`T.Kernel.dll`, `T.Library.dll`, `T.ProjectServer.dll` (already required by `T.Toolkit.dll`)	`TK_AI.cs` (atomic-path entry, intra-assembly direct call), Kernel-side TCP service host (cached-path entry, cross-assembly), `T.Designer.UI` (defaults class)

The Local AI sources are included in `T.Toolkit.csproj` via the implicit SDK glob under `FS/Toolkit/LocalAI/` — no separate project, no project reference. The Toolkit consumer (`TK_AI.cs` — the file that hosts both the canonical `AI.Execute` entry methods and the legacy `TK.AIExecute` forwarder) calls the Local AI service via an intra-assembly direct call — same DLL, same compile unit, no reflection-by-name on this binding.

Bind sites

Two callers reach the Local AI service entry methods. Both are compile-time-checked under the prescribed code organization, so signature changes surface as build errors rather than runtime nulls.

Caller	Binding	Calls
Kernel-side TCP service handler (cached path)	Cross-assembly compile-time reference from the Kernel-side service host to `T.Toolkit.dll` (the cached-path service lives in the `T.Toolkit.LocalAI` sub-namespace)	`ExecuteChatAsync (ObjectServer, clientGuid, userName, queryJson)`
Toolkit-sibling static `AI.Execute` (atomic path); legacy `TK.AIExecute` alias forwards here	Intra-assembly direct call inside `T.Toolkit.dll` (`TK_AI.cs` and the `T.Toolkit.LocalAI` service compile into the same DLL)	`ExecuteAtomicAsync (ObjectServer, queryJson)`

Type visibility

The Local AI service singleton, the cached-path orchestrator, and the atomic-path class are all `internal sealed`. The single `public static` surface is the defaults class, which exposes the recommended endpoint constants (URL, model name, etc.) for the Designer-side configuration dialog to seed its 5-field editor.

Process and thread model

Where Local AI runs

Local AI HTTP traffic is server-side only. Both paths execute their LLM POST inside the TServer process; clients never directly contact the LLM endpoint.

Caller surface	Process where LLM POST happens	Thread
`ChatRequest` Display action	TServer (TCP service handler dispatch)	TCPServer dispatch thread for the inbound ServiceClient call
`AI.Execute` from Server. Class method	TServer	Caller's thread (script execution thread); sync wrapper unwraps async via `AsyncHelpers.RunSync`
`AI.Execute` from Server. Task	ScriptTaskServer.exe (a server-domain process)	Task execution thread
`AI.Execute` from Report generator	ReportServer.exe	Report-rendering thread

Why no runtime "view-client" gate

The cached path is reachable only through the Kernel-side TCP service handler — by construction, that handler runs in the TServer process. A misrouted client-side caller cannot land in `ExecuteChatAsync`.

The atomic path is reachable from a Toolkit-sibling static, so *technically* a Display CodeBehind script (which runs at the client) could call `AI.Execute`. The platform does not block this with a runtime gate, because three platform-level safety nets already defend against degraded execution at the client:

1. **Secret resolution short-circuits off-server.** `Security.CheckAndResolveSecretCommConfiguration` returns the unresolved literal `/secret:<Name>` token when not running server-side. The LLM endpoint then receives the literal token as the credential and rejects with 401.
2. **Configuration reads route through `ObjServer.DB`.** Some client-side configurations have a stripped-config DB; reads fall back to defaults targeting `localhost:11434` — unreachable from a remote browser.
3. **Master kill-switch precedes everything.** When `SolutionCapabilities[LocalAI].Enabled` is off, the call short-circuits with `status="disabled"` before any HTTP work.

Together these mean a misrouted call from a thin client never silently succeeds with leaked credentials — it returns a normal HTTP-error envelope or the master-gate disabled envelope. No runtime view-client predicate is required.

Master gates — application order

Both paths apply gates in a fixed sequence at entry to the service method. The order is non-negotiable.

Order	Gate	Cached path	Atomic path	Outcome when gate is closed
1	`SolutionCapabilities[LocalAI].Enabled = true`	Yes	Yes	`status="disabled"`, `latencyMs=0`, no HTTP traffic
2	`SolutionSettings.ModelOptions` bit `0x02` (`EnableRuntimeMCP`)	Yes	Skipped	Cached: `status="disabled"`; atomic: not applicable
3	Per-category sub-bits (`0x04`–`0x20`)	AND-ed with master to build tool catalog	Skipped	Tool category absent from the catalog the LLM sees

The asymmetry on gate 2 is intentional. `ModelOptions` exists to gate the LLM's tool surface; the atomic path exposes no tools, so the bit is meaningless to it. Gating the atomic path on `0x02` would prevent customers from using `AI.Execute` for pure-language tasks (translation, summary, rephrase) while reserving tool-equipped chat for selected operator panels — a legitimate posture.

Cached-path mechanics

Per-Display-panel transcript cache

The cached orchestrator maintains a per-solution `ConcurrentDictionary<string, ChatSession>` keyed by `ClientInfo.Guid` — the platform's per-TCP-connection identifier (the same key shape `ModuleHistorian` and `ServiceSyncObjects` use for their own per-connection bookkeeping).

Property	Behaviour
Cache key	`ClientInfo.Guid` from the inbound TCP service call's thread-local context. NOT exposed to the runtime `Client.*` namespace.
Cache participation	Gated by `SolutionSettings.ModelOptions` bit `0x80` (`EnableChatHistory`). Default ON in new 10.1.5 solutions. When OFF, the orchestrator builds a local `SessionState` per call without entering the cache.

Login reset	Lazy. The orchestrator records the last-seen `UserName` per cache entry; when the next chat turn arrives with a different `UserName` on the same `ClientInfo.Guid`, the transcript clears before processing the new turn.
Transcript truncation	Per-cache-entry, truncated to a fixed maximum number of messages calibrated for ~8B-class context windows. Older messages drop oldest-first.
Cache lifetime	Per-process. Restarting TServer clears all transcripts. There is no persistence to disk in 10.1.5.

Tool dispatch loop

When the master tool bit is on and at least one category sub-bit is on, each chat turn assembles a tool catalog and may enter a tool-dispatch loop:

1. POST `messages` + `tools` to the LLM endpoint.
2. Inspect the assistant message. If it contains `tool_calls`, dispatch each tool in-process against the live `ObjectServer` (no HTTP loopback to RuntimeMCP — sub-millisecond direct dispatch).
3. Append tool-result messages to the running message list. Loop back to step 1.
4. Terminal exit when the assistant message is plain content (no `tool_calls`).

The loop is bounded by two limits:

- **Per-turn dispatch cap** — total tool calls per turn. **`MaxToolDispatchesPerTurn = 25` in 10.1.5.** A compile-time constant, not a configurable setting. It is a runaway-turn guard rather than the mechanism that keeps a turn inside the wall-clock budget — the headroom is there so a genuine multi-step tool chain (search, then browse, then read a value, then read history) can finish in a single turn. At the cap the worst case is 26 LLM POSTs: 25 dispatch rounds plus one terminal compose turn. On GPU-class hardware (~0.5–2.5 s per POST, the expected deployment shape) that is roughly 13–65 s; on CPU-only machines (a degraded fallback — ~2–4 tok/s, not a normal deployment) each POST takes 3–15 s, so a turn that ran to the cap would finish well past the default budget. On either hardware class it is therefore the wall-clock budget, not the dispatch cap, that ends a long turn in practice. When the cap trips, the next iteration POSTs WITHOUT new tool results so the LLM gets one final chance to compose a terminal reply, and returns `status="truncated"` if it does not.
- **Wall-clock budget** — 60 seconds by default, covering the entire loop (LLM POSTs + in-process tool dispatches combined). Configurable per solution through the Local AI model settings, clamped to 30–600 seconds.

Hook chain

`OnBeforeChat` and `OnAfterChatReply` are `event Func<string, Task<string>>` on the per-`ObjectServer` Local AI service singleton. Multicast; throwing handlers caught and logged into the reply's `warnings[]` with handler identity; chain proceeds with the un-mutated input. Hook failures do NOT flip the envelope's `status` away from `ok` — that's reserved for chain-aborting failures (LLM HTTP error, malformed query, etc.).

See Local AI Developer Reference for the reflection-attach pattern and worked hook examples.

Atomic-path mechanics

The atomic path is deliberately simple. After the master `SolutionCapabilities[LocalAI].Enabled` gate:

1. Parse query JSON (plain text wrap or structured form with `system` / `user` / `context` / `metadata` fields).
2. Read fresh `SolutionCapabilities[LocalAI].Settings` via the same defensive accessors as the cached path.
3. Resolve `/secret:<Name>` tokens via the platform's secret resolver.
4. Resolve `{Tag.X}` expressions in URL via the platform's expression resolver.
5. Decode auth multi-line via `ReportWebData.DecodeAuthorizationHeader`.
6. Build a single OpenAI-compatible `messages[]` array — no transcript prefix, no prior turns.
7. POST with empty `tools[]`, 60-second wall-clock `CancellationTokenSource`, no loop.
8. Read `choices[0].message.content`. Build the SPEC envelope (`toolTrace=[]` always for atomic).

No hooks. No transcript. No tool dispatch. No retry. Every step lives in the atomic class; nothing calls into the cached orchestrator.

Configuration topology

Surfaces (no schema migration)

Local AI uses pre-existing scaffolding: a row on `SolutionCapabilities` (the Local AI capability row carrying `Enabled` and `Settings`) plus the long-standing `SolutionSettings.ModelOptions` bitmask. No new tables; no AI-specific table.

Surface	Type	Role
`SolutionCapabilities[LocalAI].Enabled`	Boolean	Canonical master kill-switch. Off short-circuits both paths.
`SolutionCapabilities[LocalAI].Settings`	String (JSON blob)	Five-key endpoint configuration: `URL`, `Name`, `Authorization`, `Headers`, `Info`.

`SolutionSettings.ModelOptions`	Int (bitmask)	Master tool bit and per-category sub-bits. Shared bitmask with the AI Runtime Connector and the AI Designer connector. Stays on `SolutionSettings` — orthogonal to the Local AI capability switch.

Defensive accessors

Every Local AI call re-parses `SolutionCapabilities[LocalAI].Settings` JSON on every read. There is no caching layer with invalidation. The parse cost is negligible compared to the LLM round-trip; the alternative (cached parsed values + invalidation on column change) introduces a stale-cache bug surface that buys nothing measurable. Reads of NULL / empty / malformed JSON / missing keys / empty values transparently fall back to the defaults class.

Forward-compatibility on unknown keys

The Designer's 5-field editor preserves unknown JSON keys via a capture/replay mechanism, so future SPEC extensions to `Settings` (e.g., a `Provider` dialect enum) survive an open/OK round-trip in the editor without being stripped.

SecuritySecrets integration

Local AI uses the same SecuritySecrets pattern that protocol drivers, the WebData connector, the UNS tag-provider service, and the Devices module use for credentials. There is no AI-specific secret mechanism.

Aspect	How Local AI uses it
Reference syntax	`/secret:<Name>` tokens embedded in `URL`, `Authorization`, or `Headers` fields of `SolutionCapabilities[LocalAI].Settings`
Resolver	`ObjectServer.Security.CheckAndResolveSecretCommConfiguration(string)` — Kernel-side platform service
Off-server behaviour	Resolver short-circuits when not `RunningAtServer`; literal token survives. Calling code falls through to a normal HTTP-error envelope.
Encryption	SecretValue is encrypted at rest. Field is write-only via the Designer Security UI; redacted on read-back.

For the customer-facing walk-through and worked examples, see SecuritySecrets Authentication for Local AI.

Reply envelope contract

The reply envelope is a strict five-field shape, identical for both paths and identical across success / error / disabled / truncated outcomes:

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

```
{
  "text":      "<answer text or '' on non-ok status>",
  "status":    "ok | error | disabled | truncated",
  "toolTrace": [ ... ],
  "latencyMs": 1247,
  "warnings":  [ ... ]
}
```

Architectural guarantees:

- **Always parseable.** Every envelope is well-formed JSON. `JObject.Parse(reply)` never throws on a Local AI reply.
- **Field stability.** All five fields present on every envelope, every code path. Adding new top-level fields is non-breaking.
- **`latencyMs` always present** — including on errors. Pre-condition errors (gates) report `0`; post-condition errors report actual elapsed time.
- **Never throws.** Network failures, parser failures, gate-off conditions, internal exceptions — all become envelopes with `status="error"` or `status="disabled"` and a populated `warnings[]`.

Full schema and worked examples: Local AI Reply Envelope Schema.

Integration points with other platform services

Service	How Local AI uses it
ObjectServer (per-host runtime)	Local AI service is per-`ObjectServer` singleton (`LocalAIService.Get(objSrv)`); accessors route configuration reads through `objSrv.DB`.
Security (SecuritySecrets resolver)	Token resolution at call time, server-side only.
SolutionCapabilities (per-capability row)	The Local AI capability row carries the `Enabled` kill-switch and the `Settings` JSON; one row per solution, no per-capability table.

SolutionSettings (configuration)	Carries the `ModelOptions` tool-options bitmask only; the Local AI master switch and endpoint live on SolutionCapabilities.
Kernel TCPServer (cached path entry)	Hosts the `ChatProcessService` handler that the Display ChatRequest action calls via ServiceClient TCP.
Toolkit-sibling static (atomic path entry)	`AI.Execute` (in `TK_AI.cs`; legacy `TK.AIExecute` alias forwards here) reaches the Local AI service via an intra-assembly direct call — both `TK_AI.cs` and the `T.Toolkit.LocalAI` service compile into `T.Toolkit.dll`.
WebData connector	Reuses `ReportWebData.DecodeAuthorizationHeader` for auth multi-line decoding — single source of truth for the wire format.
RuntimeMCP tool catalog	Cached path's tool catalog mirrors the same `runtime_*` tool surface that `RuntimeMCP.exe` exposes to external AI clients — single tool definition, two transports (in-process for cached path, HTTP for external clients).
AsyncHelpers	Both paths use `T.Library.tRPC.AsyncHelpers.RunSync` for sync-over-async wrapping (NOT raw `.Result` / `.GetAwaiter().GetResult()`) — handles SynchronizationContext correctly and unwraps `AggregateException`.

What Local AI deliberately does NOT add

Not added in 10.1.5	Why
An `AI` runtime root in the Object Model	The Object Model roots (`Tag`, `Server`, `Client`, `Alarm`, `Device`, etc.) do not gain an `AI` root. Surface remains `AI.Execute` (script API) + `ChatRequest` (Display action). A peer of `InfoSolution.Settings` exposing AI fields is post-10.1.5.
A `T.Modules.AI` TModule	Local AI does not contribute Object Model namespace surface; it lives as a sub-namespace inside `T.Toolkit.dll`, not as a runtime Module.
New `SolutionSettings` columns	The existing surfaces (the Local AI capability row on `SolutionCapabilities`; the `ModelOptions` bitmask on `SolutionSettings`) carry everything. Schema migration would impose cross-version routing risk for no benefit.
A provider-dialect enum (OpenAI / Ollama / Anthropic)	The dialect is implicit (OpenAI-compatible JSON). Adding a discriminator waits for a real customer needing a non-conforming endpoint.
Streaming output	Sync request/response only. Each call returns one complete envelope when the model finishes.
A bundled LLM runtime or model weights	Customers install Ollama themselves and pull the model from the official registry. Keeps the FX installer lean and avoids redistribution licensing friction.
HTTP loopback to RuntimeMCP for tool dispatch	Tool dispatch is in-process direct against the live `ObjectServer` — sub-millisecond vs 10–50 ms over-the-wire to localhost. Also guarantees the cached path works in standalone deployments without RuntimeMCPHttp running.
Atomic-path hooks or atomic-path tools	Atomic is a deliberately stateless one-shot. A customer wanting hooks or tools should use the `ChatRequest` Display action surface.

Versioning and forward-compatibility expectations

- **Reply envelope schema is LOCKED.** Adding new top-level fields in future versions is non-breaking; existing five fields stay present and stay typed as documented.
- **Configuration topology is LOCKED.** The capability-row + `ModelOptions` shape will not change. Future configuration extensions land as new keys inside the `SolutionCapabilities[LocalAI].Settings` JSON or as new bits in `ModelOptions`.
- **Two-path independence is LOCKED.** Future features ship in one path or the other (or both, with separate implementations). The two paths will not be merged behind a shared helper.
- **Code organization may evolve.** The `T.Toolkit.LocalAI` sub-namespace home is stable for 10.1.5; later releases may split or rename internal types. The compile-time entry surface (`ExecuteChatAsync`, `ExecuteAtomicAsync`) is signature-stable.
- **Script API surface naming.** `AI.Execute` / `AI.ExecuteAsync` (in `T.Toolkit.LocalAI`) is the canonical script API for new code. The legacy `TK.AIExecute` / `TK.AIExecuteAsync` aliases remain callable for backward compatibility but are `[Obsolete]` (CS0618) and hidden from IntelliSense.

See also

- Local AI — the customer-facing capability page (overview + quick start).
- Local AI Configuration Reference — endpoint, master enable, and bitmask configuration.
- ChatRequest Action Reference — the operator-chat surface.
- AI.Execute API Reference — the atomic script surface (canonical `AI.Execute` + legacy alias).
- Local AI Reply Envelope Schema — the uniform reply envelope.
- SecuritySecrets Authentication for Local AI — credential management.
- Local AI Developer Reference — hook attachment, custom tools, advanced patterns.

In this section...