

AICONTEXTBUILDER

AICB - MCP Server

Describes AICB V0.5.505.1

As of 2026-10-26

Dadera · www.dadera.de · aicb@dadera.de

Contents

1 Overview and connecting a client

1.1 What the MCP server is
1.2 Starting the server: `aicb mcp`
1.3 The stdio channel
1.4 Connecting a client
1.5 Clients and protocol revisions
1.6 Server identity and handshake metadata
1.7 What your agent reads at the handshake
1.8 Unknown arguments are disclosed, not rejected

2 Setting up an agent: aicb init

2.1 What it writes at a glance
2.2 Before you run it
2.3 Options
2.4 Step by step: the first setup
2.5 The `.mcp.json` entry
2.6 The agent skills
2.7 The symbol guard
2.8 Harness detection with `--hooks auto`
2.9 What a run reports
2.10 Turning the guard off, or removing it
2.11 What this means for your agent workflow

3 Sessions and staleness

3.1 Starting a session: the solution path as a session ID
3.2 The session cache
3.3 Recalled sessions
3.4 How tools refer to a session
3.5 The staleness disclosure
3.6 Auto-refresh: Off, Reactive, Proactive
3.7 The change watcher
3.8 One refresh at a time
3.9 Staleness or incompleteness?
3.10 The `dbPath` parameter
3.11 Timing constants via environment variables

4 Profiles, pools and facets

4.1 Pools: what the server registers and what it delivers
4.2 The four built-in profiles
4.3 Selecting a profile
4.4 Overriding the tool set
4.5 The nine facets
4.6 Styles and bundles
4.7 The profile editor
4.8 Step-by-step

5 Calling tools: conventions, batch and aicb call

5.1 The four call paths
5.2 `batch`: many read-only queries in one round-trip
5.3 `measure`: size an answer before you pull it
5.4 `aicb call`: one-shot invocation without a server

5.5 Parameter aliases
5.6 Silently dropped arguments
5.7 Which tools write
5.8 Recurring conventions across the catalog

6 Tool reference: orientation, sessions and symbols

6.1 About this chapter
6.2 Orientation, operating manual and server introspection
6.3 Sessions and analysis
6.4 Finding symbols, signatures and single-symbol context

7 Tool reference: impact, hierarchy, tests and DI

7.1 Conventions used in this chapter
7.2 Fan-in, impact and paths
7.3 Hierarchy, implementations and lifecycle
7.4 Tests and coverage
7.5 Dependency injection

8 Tool reference: facts, dead code, metrics and patterns

8.1 Facts about side effects, external calls, attributes and semantics
8.2 Dead code, cycles and metrics
8.3 Patterns and documentation drift

9 Tool reference: markup, export, review and insights

9.1 Markup: XAML/AXAML bindings and resource keys
9.2 Packing and exporting context
9.3 Diff, review and change gates
9.4 Insights

10 Tool reference: snapshots, configuration, memory and wiring

10.1 Snapshots and comparison with a stored state
10.2 Solution configuration (layers, exclusions, test detection)
10.3 Cross-process memory
10.4 aicb database entities
10.5 Agent wiring
10.6 Multi-dispatch
10.7 Alphabetical index of all tools

11 Configuration, drift and usage recording

11.1 Server configuration
11.2 Telling whether the server still matches your configuration
11.3 What the server records about tool calls
11.4 Installing the symbol guard into your agent harness: `install_agent_hooks`
11.5 MCP resources

12 Troubleshooting the MCP server

12.1 The client does not list any `aicb` tools
12.2 Protocol revision and transport
12.3 `Tool '<name>' is not in the active profile`
12.4 The agent sent an argument name the tool does not have
12.5 `MCP error -32000: Connection closed`
12.6 Answers describe the code as it was before your edit (staleness)
12.7 A result looks wrong: the analysis run was incomplete
12.8 "Unknown session" and other session-resolution failures
12.9 Two analyses of the same solution at once
12.10 Database-path errors
12.11 Self-diagnosis

12.12 What the server writes to stderr at startup
12.13 Server-side guards and limits at a glance

1 Overview and connecting a client

`aicb mcp` starts the AIContextBuilder MCP server (Model Context Protocol). It makes the analysis engine available to an AI coding agent as a set of **tools**: the agent connects to the server, asks questions such as "who calls `OrderService.Submit`?", and gets its answer from a Roslyn analysis of your solution instead of from a text search.

This chapter covers what the server is, how to start it, what a client has to be told, which clients and protocol revisions are served, what identity the server reports, and exactly what your agent reads at the handshake.

1.1 What the MCP server is

- **It is not a separate program.** `aicb mcp` is a verb of the same `aicb` command you use for everything else. There is nothing extra to install for the server itself.
- **One server process per client, stdio transport.** The server speaks JSON-RPC on standard input/output. A client starts it as a child process and talks to it over that channel. When the client disconnects (stdin closes) or the process is cancelled, the server shuts down cleanly with exit code `0`.
- **Session-bound.** The server keeps an analysis of your solution warm. Most tools take a `sessionId`; every tool that takes one also accepts the absolute path of a `.sln`, `.slnx` or `.slnf` file directly and analyzes it on first use, so calling `analyze_solution` first is optional. Answers come from that cached analysis graph; after you edit `.cs` files, `refresh_session` rebuilds it and `get_diagnostics` is the fast pre-build check.
- **A curated tool pool.** The server registers its whole tool surface but exposes a curated pool. Without further configuration that is the built-in `Default` profile with **54 tools** covering all nine task facets. The built-in `Full Select` profile serves the unnarrowed **72**: start the server with `--mcp-profile mcp-profile/full` for it. The environment variable `AICB_MCP_TOOLS` overrides the pool for the process (`all`, `lean`, or a comma-separated list of tool class names) and wins over the active profile.
- **Read-only with respect to your code.** Nothing any tool writes is anything Roslyn reads, so every answer is about the tree you wrote. Tools that persist something outside the analysis - `apply_solution_config`, `export_markdown`, `install_agent_hooks`, `save_session`, `remember_codebase`, `refresh_remembered`, `import_constellation` and `diff_review` with `recordRegressions: true` - say so in their own descriptions.
- **Nothing leaves your machine.** The server has no outbound network capability at all.

1.2 Starting the server: `aicb mcp`

The verb describes itself as:

> `Start the MCP server (Model Context Protocol, stdio JSON-RPC) - for Claude Code / Cursor / Cline. With --db-path profile-aware (render slots + tool set + skill from the active MCP profile); --mcp-profile pins which profile that is, for this process only.`

You normally do not start it yourself - your MCP client starts it from its configuration. The command has exactly two options:

Option	Effect	Default
`--db-path <file>`	Binds the server to a specific `aicb` configuration database and makes it profile-aware: `export_markdown` renders through the active MCP profile, the exposed tool set follows that profile, and its working style shapes the server instructions. The database also supplies the per-solution configuration (layer, exclusions, test detection).	none - the **standard configuration database is resolved automatically**
`--mcp-profile <id>`	Pins the active MCP profile for **this process**. The pin is process-local: it writes nothing back, so a second server or the GUI keeps its own active profile. An unknown id aborts the start.	none - the profile marked active in the configuration database applies

`--db-path` and the standard configuration database

"Without `--db-path`" does **not** mean "without a database". When you omit the option, the server resolves the standard configuration database - the same one the GUI uses, on Windows `%APPDATA%\AIContextBuilder\user-data\aicb.acb` - and announces the path on stderr:

```
No --db-path given; using the standard config DB (same as the GUI): <path>
```

The database is created and migrated on first start if it does not exist, so a first-time user without a GUI starts profile-aware rather than profile-blind. Only if that resolution or the migration fails does the server start without a profile, with a warning:

```
warning: standard config DB path could not be resolved (<reason>); starting DB-free (default profile's pool).
```

An explicitly given `--db-path` behaves differently for a file that does not exist: it is treated as a user hint and is **not** created silently. You get

```
warning: --db-path not found (<path>); starting without a profile (default profile's pool).
```

and the server starts with the default profile's pool. A schema migration failure on the given database produces

```
warning: --db-path schema migration failed (<reason>); starting without a profile.
```

With a usable database the server prints the profile it actually serves, for example:

```
Active MCP profile: 'Default' (mcp-profile/default); tool set: methods:server_info,analyze_solution,...; skill:
(none).
```

The line is extended with `[pinned via --mcp-profile; not persisted]` when you pinned a profile.

Note: `aicb mcp --help` describes `--db-path` as optional with a DB-free fallback when it is omitted. In practice the standard configuration database is resolved automatically as described above, and the server starts without a profile only when that resolution or its migration fails.

`--mcp-profile`

`--mcp-profile <id>` pins the active MCP profile for the lifetime of the process. The ids are the ones `list_mcp_profiles` reports (or the `MCP Profiles` panel in the GUI); the four built-in profiles are `mcp-profile/default`, `mcp-profile/full`, `mcp-profile/refactoring-focus` and `mcp-profile/debugging-focus`. On success the server says:

```
Pinning the MCP profile to '<id>' for this process (--mcp-profile); the profile persisted in the config DB is
left unchanged.
```

Pinning at start also keeps `tools/list` stable for the lifetime of the process, which the newer protocol revision asks for. There is no runtime switch: changing the profile means restarting the server with a different id.

A pin that cannot be applied is the one case in which the server refuses to start instead of degrading to the default profile. That covers every way it can fail - no configuration database, missing file, failed migration, unknown id, hidden profile:

```
error: unknown MCP profile '<id>' (--mcp-profile). Available: <ids>.
error: --mcp-profile '<id>' could not be applied (see above). Refusing to start on a different profile.
```

The exit code is `1`. An unknown id lists the ids that would work, because over stdio you have no other way to discover them. Hidden profiles are rejected on the same grounds: they are not listed, so pinning one would be an invisible configuration.

What else the server announces at startup

All diagnostics go to stderr, never to the protocol channel. Whether you see them depends on your client; most clients keep a server's stderr in their own log. Besides the lines above, the server writes:

Message	Meaning
`warning: MSBuild could not be registered (<reason>). analyze_solution/refresh_session will fail; the remaining tools stay usable.`	The MSBuild bootstrap failed. Analysis and session tools will not work; the rest of the tool surface does.
`MCP tool curation: <tool set>; spec=<spec>; active MCP profile=<profile>; session auto-refresh=<mode>.`	Which pool is exposed right now (a static count under `AICB_MCP_TOOLS`, or the live profile count), the tool-set specification in effect, the profile, and the session auto-refresh mode (`off`, `reactive` or `proactive`).

Exit codes of the verb:

Exit code	Meaning
`0`	Clean shutdown: the client disconnected (stdin closed) or the process was cancelled.
`1`	`--mcp-profile` could not be applied. The server refuses to start on a different profile than the one you asked for.

1.3 The stdio channel

stdout **is** the JSON-RPC channel. A single stray line written to stdout - by product code, by a library, or by MSBuild or Roslyn during an analysis - puts a non-JSON line into the protocol stream. A strict client treats that as a protocol violation and tears the connection down: the server process keeps running, but every client on it is left with `Connection closed` and no way back. This is the one failure mode a stdio server cannot survive, so the server protects the channel in two layers:

- **Logging is redirected to stderr.** Every log record the server and its libraries produce goes to stderr, at every level.
- **Every other writer is redirected too.** `Console.Out` is pointed at stderr before anything else can write, so no code path in the process - including one written later - can put a byte into the protocol channel. The transport itself is unaffected: it holds its own handle on standard output. Redirecting to stderr rather than discarding is deliberate: a stray write is a defect you want to see.

Fatal background failures are reported on stderr as well, before the process goes: an unhandled exception (`FATAL: unhandled exception - the aicb MCP server is terminating: <error>`) and an unobserved task exception (`WARNING: unobserved task exception in the aicb MCP server: <error>`). The server cannot prevent such a termination, but it turns a silent death into a named one.

Note: `Connection closed` is a channel problem, not a tool crash. Ordinary tool failures do not close the connection - they come back to the client as clean error responses.

1.4 Connecting a client

The quick path: `aicb init`

Run `aicb init` in your project directory. It writes the entry your client needs into `.mcp.json`, writes the agent skill into `.claude/skills/`, and installs the symbol guard for the agent harnesses in use. Codex and OpenCode do not read `.mcp.json`: for each of them that the project uses (or that `--hooks` names), the same run also registers the server in that client's own configuration - `.codex/config.toml` or `opencode.json` - and writes the skill to `.agents/skills/`, where both read skills. It never overwrites an existing file unless you pass `--force`, so re-running is safe.

Option	Effect	Default
`--path <dir>`	Project directory to wire up.	the current directory
`--force`	Overwrite artifacts that are already there. Without it, an existing `aicb` entry or skill file is left untouched.	off
`--skills context|all`	`context` writes the `aicb-csharp-context` skill; `all` adds the `aicb-code-review` / `aicb-code-simplifier` review pair and `aicb-usage-check`.	`context`
`--hooks auto|none|all|claude-code|codex|opencode`	Which agent harnesses to install the symbol guard for: the ones already used in this project, none, all, or one by name. Naming Codex or OpenCode also registers the server for it.	`auto`

The run reports each artifact with one of four labels - `created`, `updated`, `kept`, `refused` - and ends with:

```
Restart or reconnect your MCP client, then call server_info to confirm it took.
```

That last line is the step `init` cannot do for you: a client reads its server list at startup. If you only want the server wiring and no guard, pass `--hooks none`. That writes no enforcement; it does not remove an installation that is already there, and it still registers the server for a Codex or OpenCode project.

One client-specific step stays with you, and the output names it: **Codex** reads a project's `.codex/config.toml` only in a project it trusts. Open Codex in the folder once and trust it, or the `aicb` entry stays invisible to it. The entry `init` writes for Codex carries `tool_timeout_sec = 300`, because Codex gives up on a tool call after 60 seconds by default and the first call on a large solution analyzes it; an entry written by an earlier version lacks the line - run `aicb init --force`, or add it by hand.

Writing the client entry by hand

`.mcp.json` in your project root holds the server list. The minimal entry is:

```
{
  "mcpServers": {
    "aicb": {
      "command": "aicb",
      "args": ["mcp"]
    }
  }
}
```

`command` is resolved by your client through the normal executable search. If `aicb` is not on the `PATH` your client sees, use the absolute path of the executable instead:

```
{
  "mcpServers": {
    "aicb": {
      "command": "C:\\Tools\\AICB\\aicb.exe",
      "args": ["mcp"]
    }
  }
}
```

On Windows a JSON string needs its backslashes escaped as `\\`. Any `aicb mcp` option can be placed in `args`, for example:

```
{
  "mcpServers": {
    "aicb": {
      "command": "aicb",
      "args": ["mcp", "--mcp-profile", "mcp-profile/full"]
    }
  }
}
```

Note: do not confuse `.mcp.json` (a client's server list, in your project) with `aicb.mcp.json` (the server's own settings, in your user configuration directory, on Windows `%APPDATA%\AIContextBuilder\aicb.mcp.json`). The server only reads the latter; it never writes it.

What `aicb init` does to an existing `.mcp.json`

`init` merges the `aicb` entry into the server list it finds and never disturbs the other entries. There are five possible outcomes, each with its own sentence in the report:

Outcome	Report sentence
No file was there	`created with the aicb entry`
The file was there, other entries were kept	`aicb entry added, existing entries kept`
An `aicb` entry was already there	`already has an aicb entry - left untouched`
An `aicb` entry existed and `--force` replaced it	`existing aicb entry replaced (--force)`
The file is not usable as a server list	`left untouched - not valid JSON, or a duplicate key, or the top level is not an object, or 'mcpServers' is not an object; fix or remove it and re-run`

The last row is the load-bearing rule: when the existing document does not parse, contains a duplicate key, or is not a JSON object, the command writes **nothing**. Overwriting a configuration you own is unreachable rather than merely unlikely. Parsing is strict - comments and trailing commas are not accepted - because a tolerant read followed by a write would silently delete whatever the tolerance swallowed. Without `--force`, an existing `aicb` entry is left exactly as it is; you may have pointed it at a different binary, a wrapper script or a debug build.

After connecting

1. Restart or reconnect your client so it reads the new server list.
2. Ask your agent for something concrete, for example "*find the callers of* `OrderService.Submit`".
3. Call `server_info` to confirm the connection took and to see the identity and drift report described below.

1.5 Clients and protocol revisions

The server serves both current MCP protocol revisions over stdio, from the same tool pool:

Revision	Entry point
`2026-07-28`	`server/discover` - stateless; every request carries its own metadata
`2025-11-25`	`initialize` handshake

This matters because MCP has no fall-forward: a client that speaks only the older revision has no way to reach a server that speaks only the newer one. Serving both means the client picks whichever revision it knows. Older revisions are accepted as well, because the MCP SDK the server is built on keeps its backward compatibility; the server itself does not branch on the revision at any point. Observed against a running server:

Client	`clientInfo.name`	Protocol revision observed
Claude Code	`claude-code`	`2026-07-28` and `2025-11-25`
Codex	`codex-mcp-client`	`2025-06-18`
OpenCode	`opencode`	`2025-11-25`

The verb description and the README also name Cursor and Cline; neither has been verified against a running server.

Two limits are worth stating plainly:

- **stdio only.** There is no HTTP/SSE transport, so the HTTP-specific parts of the newer revision (session headers, resumability, OAuth) do not apply here.
- **Instructions are delivered, not pushed.** The server sends them at the handshake, but whether a client ever re-reads them is the client's decision. After changing the active profile, restart the server (or reconnect the client) so the agent sees the new instructions.

Two facts matter when a profile changes while a server is running:

- A running server keeps the configuration it started with: the tool list, the instructions and the session auto-refresh mode are fixed at start. A profile edited in the GUI (or by another process writing the configuration database) reaches the server only after a restart or reconnect; until then `server_info` reports the pending change as CONFIG DRIFT.
- `tools/list` carries caching hints: a time-to-live of **15 minutes** and a private cache scope (the list must not be shared between users). The server declares both `tools/list_changed` and `resources/list_changed` but sends neither notification, so a client may keep its cached tool list until that time-to-live expires. Restart the server after changing the active profile, and reconnect the client.

1.6 Server identity and handshake metadata

The server introduces itself to every client as:

Field	Value
Server name	`aicb` - deliberately the CLI tool name, not the internal assembly name. This is the name clients and directories display.
Server version	The version of the installed binary, the same number `aicb --version` prints. It is protocol- and telemetry-bearing: it travels in the handshake and is recorded with every tool call.
Instructions	The fixed text plus the active profile's working style - see below.
Capabilities	`logging`, `resources` and `tools`; both `resources` and `tools` declare `listChanged` (the server sends neither list-changed notification).

The git commit the binary was built from is **not** part of the handshake. `server_info` reports it separately when the assembly's informational version carries one. Public installer, ZIP and dotnet-tool release builds deliberately omit it; development builds can include it. When present, it distinguishes binaries that share a version number; when absent, `server_info` does not guess.

1.7 What your agent reads at the handshake

The delivered text

The instructions are composed from ordered blocks, not from one string. The order is the delivery order, and it is deliberate: everything that changes what the agent **does** stands before everything that merely informs it, so a client that cuts the text keeps the behavior rules. The fixed text is:

```
AIContextBuilder (aicb) - a Roslyn MCP server that answers questions about a C#/.NET solution. For C# SYMBOL
questions use aicb tools, never grep/Read: e.g. find_usages, impact_of_change, find_implementations,
find_tests_for and find_symbol. Text search matches only substrings and misses overloads, aliases, partial
types and interface dispatch. Symbol-or-text self-test: SYMBOL = C# type, method, interface or member. Plain
text = literals, comments, logs/config keys; non-C# files stay with text search.
---
Pre-edit gate: before ANY edit that changes a symbol other code names - a rename included - the FIRST call is
impact_of_change; the trigger is the symbol's REACH, not the kind of edit. It returns the transitive fan-in
closure, including consumers a compile does not check (a count assertion, a non-exhaustive switch statement)
and a direct find_usages does not show.
SELF-INIT: every tool that TAKES a session also accepts the absolute .sln/.slnx/.slnf path directly and
analyzes it on first use.
analyze_solution is optional. After editing .cs files: refresh_session FIRST, then get_diagnostics as a fast
pre-build check - EVERY session-bound tool, get_diagnostics included, may otherwise answer from the stale pre-
edit graph.
On a new solution, check solution_config_status; if an axis (layer / exclusions / test) is uninitialized, run
init_solution_config then apply_solution_config (persists a git-tracked .aicb.json next to the .sln).
New to aicb? docs() is the operating manual; list_skills is the full tool map.
list_insights(sessionId) surfaces code-quality / async / design-smell findings.
The session-less tools (server_info / list_skills / usage_report / list_mcp_profiles) take no path at all, and
a comparison tool such as verify_claim takes TWO sessions, each of which accepts one.
Tool results are structured for machine reading and may include source locations, confidence or bounded follow-
up hints; use the returned evidence when reporting a conclusion.
DB-entity tools outside the default profile's pool, expose them via AICB_MCP_TOOLS: list_run_templates /
list_constellations browse the reusable run-templates and constellations in an aicb DB, and
import_constellation imports a constellation JSON into one.
```

The line breaks above follow the block boundaries for readability. On the wire the blocks are concatenated without separators; the only line break inside the fixed text is the `---` separator after the self-test. Most blocks end with a space.

What each rule tells the agent:

Rule	Purpose
Identity	What the server is. One sentence - an agent that reads nothing else still knows what it is connected to.
Symbol questions	For C# symbol questions use aicb tools, never grep or a file read. Names five examples; explains that text search misses overloads, aliases, partial types and interface dispatch.
Symbol-or-text self-test	How to decide in the moment: a symbol is a C# type, method, interface or member; literals, comments and configuration keys are plain text.
Pre-edit gate	Before any edit that changes a symbol other code names - a rename included - the first call is `impact_of_change`. The trigger is the symbol's reach, not the kind of edit.
Self-init	Every session-taking tool accepts a `.sln`, `.slnx` or `.slnf` path directly.

Rule	Purpose
Refresh after edit	`analyze_solution` is optional; after editing `.cs` files call `refresh_session` first, then `get_diagnostics` as the fast pre-build check.
Solution configuration	On a new solution, check `solution_config_status`; initialize an uninitialized axis (layer, exclusions, test) with `init_solution_config` and `apply_solution_config`.
Manuals	`docs()` is the operating manual, `list_skills` the full tool map.
Insights	`list_insights` surfaces code-quality, async and design-smell findings.
Call shapes	The session-less tools (`server_info`, `list_skills`, `usage_report`, `list_mcp_profiles`) take no path; a comparison tool such as `verify_claim` takes two sessions.
Result shapes	Results are structured for machine reading and may carry source locations, confidence or bounded follow-up hints; report the returned evidence.
DB-entity tools	`list_run_templates`, `list_constellations` and `import_constellation` are outside the default profile's pool and are exposed via `AICB_MCP_TOOLS`.

Delivery budgets

The server **never truncates** the instructions - it sends the complete composed text. Two client-side budgets shaped the order of the text:

Boundary	Value	Meaning
Codex discovery surface	first **512 characters**	Codex documents the first 512 characters as the surface available before deferred tool loading. This is a placement requirement, not a claim that Codex truncates there: the identity line and the discovery rules are placed so that this prefix is self-contained.
Claude Code prompt prefix	first **2,048 characters**	The largest prefix of the instructions known to be delivered by a client, measured against Claude Code.

Both values are **characters**, not bytes, and both protocol revisions use the same budget - the newer revision does not deliver more. The 2,048-character figure is a measurement, not a preference, and it is not a maximum length for the text: a client with a larger budget receives all of it. That is why the fixed text is ordered by delivery priority rather than narrative flow, and why the profile's working style is placed before the reference material instead of appended at the end.

The MCP profile editor in the GUI previews the composed text under the label `Exactly what the MCP server sends for this profile, shown in two client-specific availability boundaries.` and draws a cut mark at the measured budget, so you can see whether a working style lands above or below the line.

Omission under a narrowed profile

A rule that names tools is an invitation to call them, and a call the server then refuses is indistinguishable from a wrong tool choice. When the exposed pool does not contain **any** of the tools a rule names, the server therefore leaves that whole rule out of the handshake. Consequences worth knowing:

- The identity line and the discovery rules are never omitted.
- A rule is dropped only when **all** of the tools it names are unavailable; a rule that still names one reachable tool keeps its text, including the names it cannot reach.
- The block about the DB-entity tools is never omitted either, because its subject is precisely tools outside the pool; it keeps the phrase `outside the default profile's pool` **before** every name it mentions, so a client that cuts the text cannot receive a name without the reason not to call it.
- The same exposed set drives listing, callability and the instructions, so "listed", "callable" and "named in the handshake" cannot drift apart - including under the `AICB_MCP_TOOLS` override.

The working style

The active MCP profile's working style is spliced into the middle of the fixed text, after the behavior rules and before the reference material, under the heading `Working style:`. It is spliced rather than appended because an appended style would stand behind the whole fixed text - well past the measured budget, so no client would ever receive one.

The guarantee is one number: a profile with exactly **one** guidance block is delivered whole. From the second block on the style may be cut by a client with the measured budget; that is deliberate, and the profile editor discloses it instead of promising it away (`Part of this profile's working style falls below the line and is dropped by such a client. Exactly one guidance switch is guaranteed to arrive whole; beyond that it depends on their combined length, which the cut mark above shows.`).

When the instructions change

Instructions are read **once**, at the handshake; the protocol has no instructions-changed notification. A profile edit therefore reaches the text at the next server start. In between, the `server_info` tool reports the change as CONFIG DRIFT - its signal that the running server serves a different configuration than the database now holds, and that a restart is due.

1.8 Unknown arguments are disclosed, not rejected

The MCP SDK binds the arguments a tool knows and drops the rest without a word. When the mistyped name belongs to a **required** parameter, the call fails to bind and you get a visible error. When it belongs to an **optional** parameter, there is no signal at all: the call succeeds and answers a different question than the one you asked. A namespace filter that was silently dropped, for example, turns a local answer into a solution-wide one that reads like a local finding.

On a successful call

Every successful response that had argument names dropped carries an extra content block of this form:

```
<!-- ignored-arguments: <names> are not parameters of <tool> and had NO effect on this answer - it was computed as if they had not been sent. <tool> takes: <parameter list>. -->
```

The block is a remark about the request, appended next to the answer rather than spliced into it, so it cannot corrupt the payload. At most eight dropped names are spelled out; the rest are summarized as `(+N more)`. A name is reduced to the identifier characters a real parameter name could contain, and becomes `(unprintable)` when nothing is left. The server stays silent when it cannot speak with authority - for an unknown tool, or a tool whose parameter list it cannot read, no note is emitted, because a half-known inventory would accuse you of sending something invalid on the strength of a failed lookup.

The disclosure is not a refusal. The answer is correct for the call **as bound**, and refusing would turn a working call into a hard failure over what may be a stray field. What the note gives you is the visibility to fix the call. One concrete case worth remembering: the parameter for namespace filtering is named `scope`.

On a failed call

A mistyped required parameter fails before the tool is entered. The error then names the tool's real arguments instead of only reporting the failure:

```
<ExceptionType>: <cause> - This is an ARGUMENT-BINDING failure: the call never reached the tool, so retrying it unchanged fails identically. <tool> takes: <parameter list, required ones marked (required)>. You sent: <names>. Not a parameter of this tool: <names>.
```

The message states which arguments the tool takes, which were passed, and which of those are not recognized - so the fix does not require comparing your call against the input schema again. Retrying the same call unchanged is called out explicitly, because it fails identically every time.

2 Setting up an agent: aicb init

`aicb init` is the command that connects a project to an agent. It writes the files an MCP client reads - the server entry and the agent skills - and, for the agent harnesses the project uses, it installs the **symbol guard**: a hook that refuses a C# symbol search and redirects the agent to the semantic aicb tools.

The command only writes files. It opens no solution and starts no analysis: no Roslyn, no MSBuild, no database. You can run it in any project directory, before anything is built, and even in a directory that holds no solution at all - the guard itself only acts where a solution is present (see "When the guard stays silent").

2.1 What it writes at a glance

Artifact	Location, relative to the project directory	Written when
MCP client entry	`.mcp.json`	always
Server entry for Codex / OpenCode	`.codex/config.toml` / `opencode.json`	for Codex or OpenCode when detected or named, also with `--hooks none`
Agent skills	`.claude/skills/<skill>/SKILL.md`	always; one skill by default, three with `--skills all`
Agent skills for Codex / OpenCode	`.agents/skills/<skill>/SKILL.md`	for Codex or OpenCode when detected or named
Symbol guard script(s)	`.claude/hooks/`, `.codex/hooks/` or `.opencode/plugins/`	for every harness that is detected or named
Harness wiring	`.claude/settings.json`, `.codex/hooks.json` or `opencode.json`	for every harness that is detected or named

The run attempts the artifacts in a fixed order: `.mcp.json` first, then the Codex and OpenCode server entries, then the skills, then **all** guard scripts, then **all** wiring entries. Harness detection happens before the first write, so the run can never count a directory it created itself as evidence that a harness is in use. Each artifact reports its own result - one failure does not cost the others and does not hide them.

Every file is written atomically: a temporary file in the same directory is filled first and then moved over the target. An interrupted run therefore cannot leave a truncated `.mcp.json` (which would silently remove every other MCP server you had registered).

2.2 Before you run it

`aicb init` assumes the `aicb` command is already installed. The .NET tool needs the .NET 10 SDK:

```
dotnet tool install -g aicb-roslyn-mcp
aicb --version
```

On Windows with the desktop app, install the Windows installer instead - it contains the same server and keeps one `aicb` per machine. If more than one `aicb` is on your `PATH`, `aicb init` warns you (see "What a run reports").

Then open a terminal in the project directory you want to wire up:

```
cd /path/to/your/project
aicb init
```

2.3 Options

Option	Default	What it does
`--path <dir>`	the current directory	The project directory to wire up. A directory that does not exist is an error: `error: directory not found: <dir>`, exit code `1`.
`--force`	off	Overwrite artifacts that are already there. Without it, an existing aicb entry, skill file, guard script or wiring entry is left exactly as it is - which is what makes re-running the command safe.
`--skills <context|all>`	`context`	`context` writes only `aicb-csharp-context`; `all` additionally writes the `aicb-code-review` / `aicb-code-simplifier` review pair and `aicb-usage-check`.
`--hooks <auto|none|all|claude-code|codex|opencode>`	`auto`	Which harnesses get the symbol guard and its wiring. `auto` installs for the harnesses already used in this project; `none` installs no enforcement at all; `all` installs for all three; the three ids name one harness each.

An unknown `--hooks` value is rejected before the directory is even checked, because a typo in your own text does not depend on the file system:

```
error: --hooks '<value>' is not one of: auto, none, all, claude-code, codex, opencode
```

`--skills` accepts only its two values; anything else is rejected by the command-line parser before the command runs.

2.4 Step by step: the first setup

1. Install `aicb` and confirm it answers: `aicb --version`.
2. Change into your project directory and run `aicb init`. Add `--skills all` if you also want the review pair and the usage check, or `--hooks none` if you want no enforcement.
3. Read the report. Every artifact gets one line; if a line says `refused`, nothing was written for that file, the exit code is `1`, and the line says what to fix. Repair the file and run the command again.
4. Restart or reconnect your MCP client - a client reads its server list at startup, and this is the one step `aicb init` cannot do for you.
5. Call `server_info` to confirm the connection. It needs no session, no solution and no build, so it is the cheapest proof that the wiring took.
6. Start asking questions: the agent passes the absolute path of your `.sln` as the session argument (self-init) and works with the semantic tools from then on.

2.5 The `.mcp.json` entry

The file is created in the project root if it does not exist, and receives exactly this entry:

```
{
  "mcpServers": {
    "aicb": {
      "command": "aicb",
      "args": ["mcp"]
    }
  }
}
```

The `command` is the bare name of the globally installed tool, not a path, so the entry is portable. The file is written with indentation and a trailing newline, because it is hand-edited and usually kept in version control.

What happens when the file already exists:

Situation	Line printed by the command
No file was there	`created with the aicb entry`
File existed, other servers kept, aicb added	`aicb entry added, existing entries kept`
An aicb entry existed and `--force` replaced it	`existing aicb entry replaced (--force)`
An aicb entry existed, no `--force`	`already has an aicb entry - left untouched`
The file cannot be used as a server list	`left untouched - not valid JSON, or a duplicate key, or the top level is not an object, or 'mcpServers' is not an object; fix or remove it and re-run`

Notes on the refusal:

- It names all four possible causes on purpose, because three of them are valid JSON: a syntax check in your editor does not rule them out.
- The file is parsed strictly - no comments, no trailing commas. A tolerant read followed by a write would silently delete whatever the tolerance swallowed.
- Without `--force` an existing aicb entry is never touched. You may have pointed it at a wrapper script, a debug build or another binary, and silently rewriting that is the one way this command could break a working setup.
- With `--force` the entry is replaced in place, so it stays where you had it and the diff is a single hunk. Your other entries are kept either way, including their exact spelling - non-ASCII characters in unrelated lines are not rewritten.

Three files with similar names

- `.mcp.json` - the **client's** file, in your project root. The running server never reads it; only `aicb init` reads it, to add its entry.
- `aicb.mcp.json` - **aicb's own** file, optional, in your user configuration directory. It sets the exposed tool set and the session cache for headless operation. Absent is the normal case.
- `<SolutionName>.aicb.json` - the per-solution sidecar next to your `.sln`: layers, exclusions, test detection. It belongs in your repository.

2.6 The agent skills

A skill is a `SKILL.md` file with YAML front matter that an agent harness loads from `.claude/skills/<name>/SKILL.md`. Its `description` is what an agent sees before deciding whether the skill applies, so it is written as a trigger.

Skill	Written by	What it is
`aicb-csharp-context`	every run (default)	The manual for driving aicb itself, addressed to the agent
`aicb-code-review`	`--skills all`	The correctness half of a post-task review pair: finds bugs
`aicb-code-simplifier`	`--skills all`	The complexity half: finds over-engineering, not bugs
`aicb-usage-check`	`--skills all`	Reads `usage_report` at the end of a task: what this server was reached for, what went untouched, what it cost

The names carry the `aicb-` prefix deliberately: `code-review` and `code-simplifier` are already taken in a typical Claude Code installation, and a skill that silently loses to a same-named neighbor is worse than one that is absent, because nothing reports the collision.

The three are opt-in because each is an opinion about how you should work, not part of the product - if you already have a review routine, aicb's should not turn up beside it unasked, and whether you audit your own tool use is your habit to choose.

What `aicb-csharp-context` tells the agent

- **Setup**: how to install aicb and register the server.
- **Step 0 - the session**: every tool accepts the absolute `.sln` path directly as its session argument (self-init); no separate analyze step is needed, and the server reads live source, including uncommitted work.
- **Which tool answers which question**: a routing table from question to tool.
- **The pre-edit gate**: before any edit that changes a symbol other code names - a rename counts, and so do a signature, a default value or a shared enum, interface or DTO - the first call is `impact_of_change`. The trigger is how far the symbol reaches, not what kind of edit it is. The tool returns the transitive fan-in, including consumers a compile does not check: a count assertion in a test, a non-exhaustive `switch` statement, a parallel list that has to grow with yours.
- **How far to trust an answer**: a rank order of tool groups by how often they misled, with the practical rule that the less a tool is called, the more its answer is a lead rather than a fact.
- **Reading the results**: an empty result is not proof of absence; result lists are capped while the reported totals stay true.
- **After you edit**: `refresh_session` first, then `get_diagnostics`. The order is not cosmetic - `get_diagnostics` compiles the session's snapshot, so without the refresh it describes the code before your edit.
- **Symbol or text**: when plain text search is still the right tool - string literals, comments, log messages, config keys, non-C# files.
- **First run on an unfamiliar solution**: `solution_config_status`, then `init_solution_config` and `apply_solution_config` if a slot is uninitialized.
- **Limits**: C# only; opening a solution runs its MSBuild logic, so analyze only solutions you trust; the fan-in graph covers exactly two markup edge kinds.

What the review pair tells the agent

Both skills are written as a workflow. They gather the pending diff first (pushed branch, `main`, last commit, plus uncommitted changes), then check their heuristics against aicb facts before judging. If aicb is not connected, both say so explicitly: `aicb-code-review` runs a build itself rather than skipping the checks silently, while `aicb-code-simplifier` works from reading alone.

- `aicb-code-review` runs aicb pre-checks on the changed solution - `refresh_session`, `get_diagnostics`, `impact_of_change` per changed symbol, `find_tests_for` and `coverage_gaps`, concurrency checks for async changes, the markup tools for XAML changes, and `list_insights` scoped to the changed files for the pre-existing smells the diff must not worsen - and then works through bug classes: logic errors, boundaries, null handling, state and lifecycle, resource leaks, error handling, concurrency, async pitfalls, data integrity and security.
- `aicb-code-simplifier` verifies its heuristics with usage, implementation and dead-code facts, and its size claims with `symbol_metrics` and `list_insights` on the changed files, and then applies lenses such as unnecessary abstraction, premature generalization, nested control flow that could be flat, dead branches, stale comments and verbose tests.

Both accept a `--high` mode (the default, a single pass) and a `--max` mode (parallel angle-finders for higher recall).

Where the skills go

Skills are always written to `.claude/skills/`, where Claude Code reads them. When the project uses Codex or OpenCode, or `--hooks` names one of them, the same skills are also written to `.agents/skills/`: Codex reads skills from there and from `.codex/skills/`, OpenCode from there, `.claude/skills/` and `.opencode/skills/`, and both read the shared folder. OpenCode keeps one copy of a skill it finds under two of its folders.

The skill is written per project. If you want it available in every project instead of one, copy the folder `.claude/skills/aicb-csharp-context/` to the same path under your home directory.

2.7 The symbol guard

The guard is the enforcement half of the setup: the skills tell an agent what aicb is for, and the hook makes the switch happen whether or not the agent read them. aicb installs **only** this hook - no other hook or plugin of any kind.

It is a Node.js script that the harness runs before a tool call (`PreToolUse` in Claude Code and Codex, `tool.execute.before` in OpenCode). For Claude Code and Codex the wiring starts it with `node`, so Node.js has to be available on your `PATH`. The guard fails open: any error inside it results in "allow", so a broken hook can never break your tools.

The one thing it refuses

A **Grep** call whose pattern is a single C# symbol - a PascalCase identifier of three or more characters (optionally starting with `I`), or a `Type.Member` pair - is denied, unless one of the two exits below applies. The agent does not get search results; it gets this message:

```
aicb-guard: this is a C# SYMBOL query (pattern='<pattern>'). grep misses overloads,
aliasing, partial types and interface dispatch - use the semantic MCP tools instead:
mcp__aicb__find_usages (who uses X) / find_symbol (where is X) / explain_symbol (what is
X) / calls_external (external BCL/NuGet member) / impact_of_change (blast radius). No
separate analyze step needed: pass the .sln directly as sessionId (sessionId="<abs path
to .sln>") and the tool self-initializes. If this is genuinely a TEXT search (string
literal / comment / log / config), restrict it to the file type (Grep glob e.g. "*.xaml"
or "*.md") or open the specific file with Read. If an aicb answer about this name does
not cover your question, this Grep is allowed once that aicb call is made.
```

The last sentence appears in Claude Code only, the one harness whose session record the guard can read.

This is the only hard block, and it has two exits (see below). A shell command is never hard-blocked, because a shell command can do anything - `Bash` calls that look like a symbol sweep only get a note.

Note: the message names the tools in the `mcp__aicb__...` form that Claude Code uses. Other harnesses may present the same tools without that prefix; the agent should use whatever form its own tool list shows.

Two ways past the refusal

- **After an aicb call on the same name** (Claude Code): when the session - the main agent or one of its subagents - has already called an aicb tool with the symbol in its arguments, the Grep is allowed with the note `aicb-guard: '<pattern>' allowed - this session already asked aicb about it (<tool>). A text search on top is fine when that answer did not cover the question. ...`. A failed call counts; a session or file path among the arguments does not. The exit is for the agent that asked aicb first and needs more than the answer gave.
- **Where aicb is not available** (every harness): when the session record shows that aicb's tools were removed or its server failed, or, without such a record, when no configuration file the harness reads mentions aicb at all, the Grep is allowed with the note `aicb-guard: '<pattern>' allowed - aicb is not available here (<reason>), so a text search is the only search there is. ...`. Any mention of aicb in those files counts as available, so in doubt the guard refuses.

What it only warns about

What the agent does	What the guard does
`Grep` with a multi-name alternation (`A|B|C`), every branch symbol-shaped	Allows, with a note: no MCP tool accepts a multi-name pattern, and the one-call semantic form is `batch` with several `find_usages` sub-queries
`Grep` with a mixed alternation (symbol-shaped and plain-text branches)	Allows, with a note naming the symbol branches

What the agent does	What the guard does
`Grep` on a single symbol after an aicb call on that name, or where aicb is not available	Allows, with a note naming the reason (see "Two ways past the refusal")
`Bash` with `grep`, `rg` or `findstr` on a symbol-shaped pattern	Allows, with a note preferring the semantic tools
First `Edit`, `Write` or `MultiEdit` on a `.cs` file in a session	Allows, with a reminder to run `impact_of_change` first and `refresh_session` + `get_diagnostics` after
Codex `apply_patch` touching a `.cs` file	The same reminder, once per session (shared with the edit reminder)
`Read` of a file of 50 KB or more without `offset` / `limit`	Allows, with a note to read a slice instead - once per session
`Read` of a file of 100 KB or more without `offset` / `limit`	The same note, every time
`dotnet build` or `dotnet test`	Allows, with a note that `refresh_session` + `get_diagnostics` pre-checks compile errors without a build - once per session

Everything except the refusal is a non-blocking reminder. Only the refusal prevents anything; the reminders cost a little context and nothing else.

When the guard stays silent

- **No `*.sln` directly in the project directory** - the guard is completely quiet in a non-.NET project or in a directory that only contains sub-projects.
- **A search scoped away from C#** - a `glob` whose every piece ends in a literal non-C# extension (for example `*.xaml`, `*.md`, `*.mjs`), a non-C# `type`, or a `path` that points at one concrete file. A negated glob (`!*.md`), a wildcard in the extension (`*.c?`, `*.verified.*`) and a glob list that includes `*.cs` still count as a search over C# files. A file-scoped search is a targeted text scan, not a codebase-wide symbol sweep.
- **A pattern with regex structure** - groups, character classes, quantifier braces (`()[]{}`) or `=>`. Such a pattern is a text search, never a bare symbol query.
- **Plain text by shape** - all-lowercase or snake_case names, UPPER_SNAKE constant or config keys, phrases containing spaces, paths and version strings are not symbol-shaped and pass through.

If you are blocked on a genuine text search, the same two routes work: restrict the search to a file type, or open the file with `Read`.

Marker files

To keep the once-per-session notes from repeating on every call, the guard writes small marker files:

- Location: `<project>/<harness directory>/cache/nudge-<session>-<key>` - for example `.claude/cache/` for Claude Code, `.codex/cache/` for Codex, `.opencode/cache/` for OpenCode.
- Keys: `edit-cs`, `read-big`, `dotnet-build`.

The harness directory is derived from the guard's own location, so a guard installed for OpenCode never creates a `.claude/` directory in a project that has never run Claude Code.

Note: these marker files are harmless and can be deleted at any time; the guard recreates them as needed. `aicb init` does not add a `.gitignore` entry for the folder, so add one yourself if you do not want them in version control.

Installing it for a harness deliberately

`--hooks auto` installs for the harnesses it detects. To install for a specific one regardless of detection, name it: `--hooks claude-code`, `--hooks codex`, `--hooks opencode`, or `--hooks all` for all three. To install nothing, use `--hooks none` - the rest of the run (the `.mcp.json` entry and the skills) still happens.

2.8 Harness detection with `--hooks auto`

`auto` is the default because writing enforcement for a harness a project does not use is not neutral: it drops configuration files into a repository you would then have to explain.

For Claude Code, a `CLAUDE.md` file in the project directory is evidence by itself: `init` never writes one, so its presence shows the project's own use. Otherwise detection asks whether a **marker directory** exists in the project - `.claude/`, `.codex/` or `.opencode/`. A marker counts as evidence when it contains at least one entry that `aicb init` itself would not have written there, because `init` creates `.claude/skills/` on every run and must not read its own side effect as proof that Claude Code is in use. Two boundary cases are deliberate:

- An **empty** marker directory counts as evidence.
- A non-empty marker counts only when it has at least one child that aicb does not create itself. For `.codex/`, `hooks/` and `cache/` are self-created; for `.opencode/`, `plugins/` and `cache/` are self-created. A marker containing only those children therefore does not count.

The price of this rule is a false negative in the safe direction: if your `.claude/` directory contains only your own `skills/` folder and the project directory has no `CLAUDE.md`, `auto` does not recognize Claude Code. You then get the explicit line

```
No agent harness detected here, so no symbol guard was installed and this run registered the server only in
.mcp.json (read by Claude Code). Pass --hooks claude-code|codex|opencode (or --hooks all) to wire one anyway -
for Codex and OpenCode that also registers the server in their own config.
```

and install it yourself with `--hooks claude-code`. Nothing is installed unasked.

Only the project directory itself is checked for `CLAUDE.md`; a file in a parent folder or a `CLAUDE.local.md` does not count. Codex and OpenCode have no such file (`AGENTS.md` is read by several harnesses and identifies none of them). The reverse case: a project that keeps a `CLAUDE.md` for another agent also gets the Claude Code guard; `--hooks` with an explicit harness, or `--hooks none`, overrides the detection.

2.9 What a run reports

Each artifact gets one line in the format `<label> <path> - <detail>`, with the labels padded to equal width:

Label	Meaning
`created`	The file did not exist and was written
`updated`	An existing file was changed (an added or replaced entry, or a `--force` replacement)
`kept`	The file was already in the desired state and was left untouched
`refused`	The file was not touched, because touching it could have destroyed something

A sample run in a Claude Code project:

```
created  C:\work\MyApp\.mcp.json - created with the aicb entry
created  C:\work\MyApp\.claude\skills\aicb-csharp-context\SKILL.md - agent skill aicb-csharp-context written
created  C:\work\MyApp\.claude\hooks\aicb-symbol-guard.mjs - aicb symbol guard written
created  C:\work\MyApp\.claude\settings.json - Claude Code: created with the aicb symbol guard wired

The symbol guard is installed and it BLOCKS: a C# symbol search is refused and redirected to the aicb tools,
not merely discouraged. Worth knowing before it surprises you.
  Remove it for Claude Code: delete the aicb entry from C:\work\MyApp\.claude\settings.json
  Skip it next time: aicb init --hooks none (that writes no enforcement - it does not remove what is already
there).

Restart or reconnect your MCP client, then call server_info to confirm it took.
```

The paragraph after the file list appears only when the run actually installed a guard. It says three things: that the guard blocks rather than discourages, how to remove it (naming the actual wiring file), and that `--hooks none` governs the **next** run rather than undoing an installation.

In a Codex project the run also registers the server and writes the skill a second time:

```
created  C:\work\MyApp\.mcp.json - created with the aicb entry
created  C:\work\MyApp\.codex\config.toml - Codex CLI server list: created with the aicb entry
created  C:\work\MyApp\.claude\skills\aicb-csharp-context\SKILL.md - agent skill aicb-csharp-context written
created  C:\work\MyApp\.agents\skills\aicb-csharp-context\SKILL.md - agent skill aicb-csharp-context written
created  C:\work\MyApp\.codex\hooks\aicb-symbol-guard.mjs - aicb symbol guard written
created  C:\work\MyApp\.codex\hooks.json - Codex CLI: created with the aicb symbol guard wired
```

and ends, after the guard paragraph, with the step only you can take:

```
Codex reads .codex/config.toml only in a project it trusts. Open Codex in this folder once and trust it, or the
aicb entry stays invisible to it.
```

For OpenCode the server entry goes into `opencode.json` (`OpenCode server list: created with the aicb entry`), the file that also carries its guard wiring.

If more than one `aicb` is on your `PATH`, every run also prints a warning, because each MCP client starts the **first** one and updating another one changes nothing a client runs:

```
warning: "aicb" is installed 2 times. Every MCP client starts the FIRST one; updating another one changes
nothing a client runs.
  runs    C:\Program Files\AIContextBuilder\cli\aicb.exe - Windows installer (AI Context Builder)
  ignored C:\Users\you\.dotnet\tools\aicb.exe - .NET tool (dotnet tool install -g aicb-roslyn-mcp)
  Keep one. With the desktop app: keep the installer and run 'dotnet tool uninstall -g aicb-roslyn-mcp'
(installed before 0.5.465.11: 'dotnet tool uninstall -g AIContextBuilder'). Without it: uninstall "AI Context
Builder" in Windows Settings > Apps.
```

The run ends with the step it cannot take for you:

```
Restart or reconnect your MCP client, then call server_info to confirm it took.
```

Exit codes: `0` on success, `1` if `--hooks` was invalid, the target directory did not exist, or at least one artifact was refused (`error: nothing was written for at least one artefact (see above).`).

2.10 Turning the guard off, or removing it

- **Install nothing next time**: `aicb init --hooks none`. This writes no enforcement, but it does not remove an installation that is already there.
- **Remove an installed guard**: delete the aicb entry from your harness's own configuration file - `.claude/settings.json`, `.codex/hooks.json` or `opencode.json`. For Claude Code and Codex this is the `PreToolUse` entry whose command names `aicb-symbol-guard.mjs`; for OpenCode it is the plugin entry `./.opencode/plugins/aicb-symbol-guard.ts`. Every run that installs a guard prints the path of the file to edit.
- **Remove the files as well** (optional): the guard script(s) under `.claude/hooks/`, `.codex/hooks/` or `.opencode/plugins/`, and the `cache/` marker directory.
- **Keep an edited guard**: an existing guard script is never overwritten without `--force`, exactly because a customized exemption list is the most likely thing to lose. The same holds for an existing wiring entry.

2.11 What this means for your agent workflow

After a default `aicb init` in a Claude Code project, the following changes for the agent working in that project:

1. A `Grep` search for a C# symbol name is **refused**, and the agent is redirected to the aicb tool that answers the question properly. Once the agent has asked aicb about that name in the session, or where aicb is not available, the search goes through with a note.
2. An `Edit`, `Write` or `MultiEdit` on a `.cs` file produces a one-time-per-session reminder to run `impact_of_change` before the change.
3. A `Read` of a file of 50 KB or more without a slice produces a reminder once per session; at 100 KB or more, every time.
4. `dotnet build` or `dotnet test` produces a one-time-per-session reminder about the faster diagnostics path.
5. The project gains a `cache/` folder with marker files under the harness directory.
6. The agent has the `aicb-csharp-context` skill available, whose description makes it load on any C# symbol question.

Only the first point prevents something. The others cost a little context and nothing else. The files land in your project, so you can commit them to share the setup with everyone working on the repository.

What the server tells your agent

Independently of the installation, the server reports the wiring state to the calling agent. The report exists only when there is something to say: when the harness is wired, the field is absent and the answer is byte-for-byte unchanged.

- **Wired**: no hint.
- **Recognized harness, not wired**: the agent is told the project does not wire the guard for its harness, that the guard turns the rule from advice into enforcement, and how to install it - with `aicb init --hooks <id>` or the `install_agent_hooks` tool - followed by the rule in words.
- **Unrecognized client, or a client that announced no name**: the agent is told that aicb ships wirings for Claude Code, Codex CLI and OpenCode and has none for it, and it is given the rule in words. On such a harness the sentence itself is the mechanism.

The hint travels as a JSON field `agentWiring` on the session information and as a leading Markdown comment `<!-- aicb agent wiring: ... -->` on raw-Markdown answers such as `get_context` and `architecture_overview`, which cannot carry a JSON field.

Installing from inside an agent: `install_agent_hooks`

The MCP tool `install_agent_hooks` is the second entry point for the guard. It writes **only** the guard script and the wiring entry - no `.mcp.json` and no skills - and it is the one aicb tool that writes into your project's agent configuration on request.

Parameter	Required	Meaning
`sessionId`	yes	A session id, or the absolute `.sln` / `.slnx` / `.slnf` path (self-init)

Parameter	Required	Meaning
`harness`	no	`claude-code`, `codex` or `opencode`. Omit it to use the harness the calling MCP client belongs to. An unknown name, or an unnamed client with no `harness` argument, is refused with the list of known ids
`force`	no	Overwrite an existing guard script and rewrite an existing wiring entry (default `false`)

There is no path parameter: the target directory is the one holding the solution the `sessionId` resolves to. The response reports the harness, the project directory, whether the run succeeded, one entry per artifact (path, outcome, detail), and the step the caller has to take next:

```
Restart or reconnect the MCP client so it loads the new wiring, then call refresh_session to clear the agentWiring hint on this session.
```

The two staleness windows it names are different: the client reads its configuration at startup, and the session keeps its analyze-time answer until it is refreshed. Installing hooks writes no source file, so a `refresh_session` alone is enough to clear the hint - no re-analysis is needed.

Note: `install_agent_hooks` is not available through `batch` or `measure`, so a call an agent believes is a read can never perform an installation.

The built-in manual

The server also carries its own manual as the `docs` tool. `docs()` without arguments returns the directory with a token estimate per page; `docs("install")` returns the page that mirrors this chapter from inside the agent, and the route `docs("init")` returns the whole first-run sequence in reading order. It is a useful answer when an agent asks how aicb is set up in a project it cannot see.

3 Sessions and staleness

A **session** is one analyzed solution that the MCP server holds in memory. Every tool that answers questions about code reads from that model; it is produced by a full analysis of the solution and stays in the server process for as long as the session lives. Because you keep editing while the server keeps answering, the model can fall behind the files on disk. The server therefore tracks how old each answer's model is, discloses it on the response, and can repair it - automatically or on request.

This chapter covers how sessions are created and identified, how they expire, how staleness is disclosed and repaired, how source staleness differs from reference incompleteness, and which timing constants you can tune.

3.1 Starting a session: the solution path as a session ID

Most tools that need a session declare a `sessionId` parameter, and that parameter accepts **two forms**. The two-session comparison tools name their sides instead: `semantic_diff` uses `sessionA` / `sessionB`, `diff_review` uses `sessionBefore` / `sessionAfter`, and `verify_claim` uses `baselineSessionId` / `changedSessionId`; each side accepts the same two forms.

Form	Example	Behavior
A session ID	the string `analyze_solution` returned	Uses that cached session.
An absolute solution path	`C:\work\MyApp\MyApp.sln`	**Self-init**: the server analyzes the solution on first access and caches it.

Self-init means `analyze_solution` is **optional**: you can pass the solution path to any session-bound tool directly, and the first call performs the analysis. The recognized extensions are `.sln`, `.slnx` and `.slnf` (a solution filter is opened like a solution). A session ID has no such extension, so the two forms can never be confused.

Concurrent tool calls that pass the **same** solution path are serialized: the solution is analyzed once, and the other callers wait for that one analysis instead of starting their own. A waiter is not left hanging indefinitely. If the wait exceeds the load-gate limit (default 45 seconds, see "Timing constants via environment variables"), the call is refused with an error that names which analysis is running and how long it has been running, states that the call never started because it was queued, and suggests waiting for the running analysis or using a smaller solution or a `.slnf` filter.

If a session reference does not resolve, the error names the cause it actually has. The three cases are distinguished:

What you passed	Message
Something with no solution extension	`'...' is neither a known session_id nor an absolute .sln/.slnx/.slnf path.` - if you meant a session ID, it is unknown or expired; call `analyze_solution` first.
A solution extension, but no file at that path	`No solution file at '...'.` - the path must be **absolute**. The message states explicitly that this is **not** an expired session.
An existing solution file	This host cannot self-initialize from a path; call `analyze_solution` and pass the returned ID.

3.2 The session cache

Sessions live in one cache per server process. Each session pins a warm workspace and the analyzed graph, so the cache is deliberately bounded:

Setting	Default	Meaning
Time to live	**90 minutes**	Sliding from the last access: every tool call that uses the session resets the clock.

Setting	Default	Meaning
Maximum sessions	**8**	Hard memory bound. On overflow the least recently used session is evicted; the session just created is never the one evicted.
Sweep interval	**5 minutes**	A background sweep releases expired sessions even when no tool call touches them.

Expiry or eviction releases the workspace behind the session. The next call with the same solution path (or a new `analyze_solution`) analyzes the solution again from scratch.

Within one server process, sessions are identified by their **solution path** (case-insensitive). If several cached sessions exist for the same path, a live one is preferred over a recalled one, and otherwise the most recently used one. A second server process has its own cache and analyzes again.

Two tools let you inspect the cache:

- `list_sessions` - all currently cached sessions that have not expired, each with its ID, solution path, project count and last access. It takes no arguments.
- `inspect_session` - the metadata of one session, addressed by its `sessionId` (solution path, project count, file-set hash, last access).

Configuring the session cache

The cache parameters can be changed without rebuilding, in two stages that are applied at server start. The cascade is **built-in defaults** < `aicb.mcp.json` < **environment variable**.

The central server configuration file is `aicb.mcp.json` in `<ApplicationData>/AIContextBuilder/` - on Windows `%APPDATA%\AIContextBuilder\aicb.mcp.json`. It is read-only for the server: you edit it by hand, and the server reads it at start. The file is hand-written JSON, so property names are case-insensitive and comments and trailing commas are tolerated. A missing file means "no override". A broken file is reported as a warning and the server starts on the defaults - so a file you believe is in effect can silently not be.

```
{
  "sessionCache": {
    "ttlMinutes": 90,
    "maxSessions": 8,
    "sweepMinutes": 5
  }
}
```

Only the fields you set override anything; unset or non-positive values keep their default.

The environment variable `AICB_MCP_SESSION` overrides the same three values for one server process. It takes a comma-separated `key=value` list:

```
AICB_MCP_SESSION=maxSessions=4,ttlMinutes=120
```

Key	Meaning	Unit
`ttlMinutes`	session time to live	minutes
`maxSessions`	maximum number of cached sessions	count
`sweepMinutes`	interval of the background sweep	minutes

Keys you do not set keep their default; unrecognized keys and values that are not a positive integer are ignored.

3.3 Recalled sessions

The memory tools persist an analyzed model in an aicb database and bring it back later - even in another process, and without running Roslyn:

Tool	What it does
`remember_codebase`	Analyzes a solution and persists the model as a snapshot. Returns a **live** session ID you can use immediately. Requires `solutionPath` and `dbPath`.
`list_remembered`	Lists the codebases remembered in a database (latest snapshot time, file-set hash, whether a usable model payload is present). Requires `dbPath`.
`recall_codebase`	Rehydrates the latest snapshot **without Roslyn** and returns a recalled session ID. Requires `solutionPath` and `dbPath`.
`refresh_remembered`	Re-analyzes a recalled session live, restores line numbers and full insights, and re-persists the snapshot. Requires `sessionId` and `dbPath`.

A recalled session behaves differently from a live one, and the differences are visible in its metadata:

- It has **no live workspace**; its origin is reported as `Recalled` (a live session reports `Live`).
- `lineNumbersAvailable` is `false`, and insights that depend on line numbers are degraded.
- The persisted payload does not carry the layer profile; a live analysis (`refresh_remembered`, or a fresh `analyze_solution`) produces a fully live session.
- Time to live and eviction work exactly as for live sessions.
- Tools that need the live Roslyn workspace reject a recalled session with a message pointing to `refresh_remembered` (or `analyze_solution`). This includes `refresh_session` - a recalled session cannot be refreshed by it.
- Snapshots written by older versions may carry no file-set hash. The staleness check then reports "unreliable" rather than "current" and stays silent (see "The staleness disclosure").
- Facts a snapshot did not record - the `.cshtml` view census, the generated-code censuses, the creations and effects of top-level statements - are disclosed rather than read as "nothing missing": the affected answers carry a `... NOT KNOWN` note that names `refresh_remembered` (see "Recall safety" under "Recurring conventions across the catalog").

`recall_codebase` also tells you whether the snapshot still matches the disk: it reports `stale: true` when the file set drifted, or the payload schema or the analyzer identity differs, and recommends `refresh_remembered` for a live re-analysis. The recalled model is served either way - that is the point of recalling.

3.4 How tools refer to a session

The tool surface uses five shapes:

Shape	Tools	Notes
One session	the majority of tools	the parameter is named `sessionId` and accepts both forms above
Two sessions	`semantic_diff` (`sessionA`, `sessionB`), `diff_review` (`sessionBefore`, `sessionAfter`), `verify_claim` (`baselineSessionId`, `changedSessionId`)	each side accepts self-init
One session plus a stored snapshot	`compare_with_previous`, `diff_public_contract`	the session is the "after" side; the baseline is a snapshot saved earlier with `save_session`
No session	`server_info`, `list_skills`, `list_mcp_profiles`, `usage_report`, `docs`, `list_sessions`, `list_insight_producers` and the database-browsing tools	take no solution argument

Shape	Tools	Notes
A path instead of a session	`analyze_solution` and `remember_codebase` take `solutionPath`; `recall_codebase` takes `solutionPath` plus `dbPath`; `compare_public_api` takes two DLL paths	

3.5 The staleness disclosure

Every response of a session-bound tool can carry a disclosure of how old the model behind it is. This matters because a "0 callers" answer out of a pre-edit graph looks exactly like a real zero, and a `find_usages` answer from a graph that is 40 minutes old is otherwise indistinguishable from a fresh one.

The check compares the **source files on disk** with the files the session was analyzed from and, for recalled state, also checks the persisted analyzer identity against the running analyzer. It has three states:

State	Meaning
`Unknown`	No usable answer (for example an unreadable solution folder). Kept separate from "current" on purpose: "I do not know" must not be reported as "changed".
`Current`	The file set on disk matches the one the session was analyzed from.
`Stale`	Source files under the solution folder have moved since analysis, or the recalled session's analyzer identity differs from the running analyzer.

Only `Stale` produces text, and the response is otherwise left byte for byte unchanged. An "everything is fine" note is deliberately never emitted: the check can only prove currency at the moment it last looked, not at the moment it answers.

The disclosure has two grammars, one per response shape:

- **JSON responses** get a `staleness` member spliced into the object. A top-level JSON array (for example `list_insights`) stays the array the tool returned; the disclosure follows it as a text block of its own, `{"staleness": {...}}`. A client that joins all text blocks of an answer and parses them as one JSON document must read the blocks separately.
- **Markdown responses** get a leading `<!-- staleness: ... -->` comment, so the caveat is visible before the document rather than after it.

A plain stale disclosure looks like this:

```
"staleness": {
  "stale": true,
  "checkedAtUtc": "2026-09-21T10:12:44.1234567Z",
  "hint": "Source files changed since this session was analyzed, so this answer comes from the pre-edit graph -
call refresh_session for a current one."
}
```

After a successful automatic re-analysis (see "Auto-refresh: Off, Reactive, Proactive") the member says so instead, and the answer is current:

```
"staleness": {
  "stale": false,
  "autoRefreshed": true,
  "mode": "full-reload",
  "hint": "Source files had changed, so this session was re-analyzed automatically (full reload) before this
call was answered - this answer reflects the code as it is now."
}
```

If the automatic re-analysis failed, the member names the failure and the answer still comes from the old graph:

```
"staleness": {
  "stale": true,
  "autoRefreshFailed": "IOException",
  "checkedAtUtc": "2026-09-21T10:12:44.1234567Z",
  "hint": "Source files changed since this session was analyzed and the automatic re-analysis FAILED
(IOException), so this answer comes from the pre-edit graph - call refresh_session to retry it explicitly."
}
```

The fields are:

Field	Meaning
`stale`	`true` when the answer comes from a graph behind the disk, `false` when the automatic path repaired it.
`autoRefreshed`	Present and `true` when the server re-analyzed before answering.
`mode`	Which path the re-analysis took: `incremental` (document texts replayed, or C# files added or removed in place, no workspace reload) or `full-reload`. Only present with `autoRefreshed`.
`autoRefreshFailed`	The exception **type** that ended a failed automatic attempt (not a message).
`checkedAtUtc`	When the check last looked at the disk. Absent after a successful automatic re-analysis.
`hint`	The same sentence as the Markdown comment: what happened and what to do.

Notes:

- The check is throttled (default 5 seconds, see "Timing constants via environment variables"). A `Stale` verdict is re-served without reading the disk again until a fresh reading supersedes it. `checkedAtUtc` therefore names the last look, not the moment of the change; the disclosure deliberately does not claim a "changed since" time, because a file-set fingerprint knows *that* something moved, never *when*.
- A disclosure is added to any call that carries an argument whose name contains `session` and that resolves to a cached session. This includes the two-session comparison tools; there, the message names which side is stale (`the session passed as sessionA, sessionB`).
- `refresh_session` is not exempt: the check runs **after** the tool, so a successful refresh classifies as current and produces no note - while a file written *during* the re-analysis is still disclosed.
- A result that is an error is not decorated; its age is irrelevant and a note would only crowd out the failure.
- A recalled session gets a different remedy in the same place: the note points to `refresh_remembered` instead of `refresh_session`, because `refresh_session` rejects a recalled session.
- `get_diagnostics` additionally leads its answer with `verdict: "stale"` and a `verdictReason` when it computed from a graph known to be behind the disk - including a project whose restore rewrote its `project.assets.json` after the session was loaded - placed before the counts so that `errorCount: 0` is not read as a clean bill.

Never infer freshness from which tool you called. Read the disclosure that the response carries.

3.6 Auto-refresh: Off, Reactive, Proactive

The auto-refresh mode decides **who pays for the re-analysis** when a session has drifted. Detection runs in all three modes; what differs is whether anything acts on it.

Mode	Who pays	Behavior
`Off`	nobody	Drift is disclosed and left in place. Repair happens only when you call `refresh_session` yourself.

Mode	Who pays	Behavior
`Reactive`	the caller	Drift is noticed when a read tool is about to answer; the re-analysis runs first and the answer comes from the current graph. The caller waits for the re-analysis (seconds on a large solution).
`Proactive`	a background watcher	A file watcher starts the re-analysis once saving has gone quiet, so the next tool call usually finds a current graph and pays nothing.

The mode is a property of the **active MCP profile** and is set in the profile editor in the GUI. The shipped rule is:

- `Reactive` **is what the normal creation paths produce.** Every built-in profile and a custom profile created with **Add New** carries `Reactive`. The GUI's **Duplicate** action does not copy this field; its clone therefore resolves to `Off` unless you set it.
- `Off` **is what an absent field means for custom/imported profiles and overridden or hidden built-ins.** A non-overridden, non-hidden built-in row is self-healed on the next load to the shipped value, which is `Reactive`; other old or imported profiles without the field resolve to `Off`.

A change to the mode takes effect at the next server start; reconnect or restart the server after editing the profile.

For one server process, the environment variable `AICB_MCP_AUTO_REFRESH` overrides the profile. The cascade is **built-in default < profile < environment variable**, resolved once at server start.

Value	Mode
`off`, `0`, `false`, `no`	`Off`
`reactive`	`Reactive`
`proactive`, `1`, `on`, `true`, `yes`	`Proactive`

Notes:

- An unrecognized value is **ignored**: the profile's value stands. A typo can therefore never switch a mode on - and never switch the shipped mode off.
- The truthy legacy spellings (`true`, `1`, `on`, `yes`) mean `Proactive`, not `Reactive`.
- The mode does not change which tools are exposed. Only `refresh_session` changes character: under `Reactive` and `Proactive` it is usually redundant, but calling it remains correct in every mode.
- There is no timeout cap on a refresh. A call may take as long as the re-analysis needs; the alternative - a cap on an estimated duration - would abort work that was about to succeed.

What the automatic path does and does not touch:

- It acts only on a session that has actually drifted **and** has a live workspace. A recalled session is skipped, because no refresh could succeed on it; the disclosure then names `refresh_remembered` as the route.
- Tools that perform their own analysis (such as `refresh_session`) are exempt, so a refresh still reports your edit as its own finding instead of being pre-empted.
- On the two-session comparison tools, the **worst** outcome is reported: if one side could not be refreshed, the note says so, even if the other side succeeded.
- The automatic path never forces. It fires on drift the check detected, so there is nothing to override; `force` is the manual escape hatch.
- **Fail-open, always.** Every failure path leaves the tool call intact and answers from the old graph, with a disclosure that says the automatic re-analysis failed and names the exception type. An automatic mode that made calls fail harder than the manual route would be worse than the drift it removes.
- After a successful automatic re-analysis the disclosure is emitted even though the answer is now current (`stale: false`), because a wait the caller did not ask for should not be silent.

3.7 The change watcher

The proactive mode adds a file watcher on the solution folder (including subdirectories). It observes C# saves and raises **one** signal once writing has stopped.

- **Quiet period (debounce): 15 seconds**, overridable with `AICB_MCP_DEBOUNCE_MS`. Every further save restarts the window, so a burst of edits produces one re-analysis rather than one per keystroke.
- **Only `.cs` files trigger it.** A change to a `.csproj`, `.xaml`, `.cshtml` or `.razor` file is not a proactive trigger; it is still caught at the next tool call.
- **The watcher is a trigger, never the source of truth.** File watchers can drop events under load (a checkout, a build, a stash). A dropped or overflowed event is treated as an ordinary signal - "something moved" - and the file-set comparison decides whether anything is actually stale. A missed event can therefore delay a refresh, never hide one.
- Watchers are armed **after** the tool call, for the sessions that call touched. Arming earlier would skip the call that creates a session.
- **Failure is silent and falls back to the reactive path.** If the watcher cannot be created (an unreachable folder, a platform watch limit), or an event is lost, the drift is still found at the next tool call and repaired there under `Reactive` / `Proactive`. The visible symptom is simply that the staleness disclosure appears again - a working watcher is the case where no disclosure appears at all.

3.8 One refresh at a time

A session has at most **one** refresh in flight. `refresh_session` and the automatic path share the same election: a call that arrives while a background refresh is already running **joins** it instead of starting a second re-analysis of the same session. The worst case for the caller is a partial wait, not a doubled one.

`refresh_session` takes:

Parameter	Default	Meaning
`sessionId`	required	the session ID (or a solution path, via self-init)
`force`	`false`	Re-analyze even when no source file has changed.

It returns `{ changed, reason, session, mode }`:

- `changed` - `false` when nothing was re-analyzed; the `reason` then says why (the file-set hash was unchanged, or every file that moved was touched without a content change).
- `mode` - `incremental` when document texts were replayed, or C# files added or removed in place, without a workspace reload, otherwise `full-reload`. It is **absent** when nothing was re-analyzed.
- `session` - the session metadata after the refresh.

Notes:

- Calling `refresh_session` after editing is cheap to do speculatively: an unchanged file set skips the expensive re-analysis. A file whose timestamp moved while its text is still the analyzed text (a save without an edit) does not count as a change either.
- The incremental path can stop engaging without any visible symptom - answers stay correct, calls just get slow again. The `mode` field is how you notice.
- `force: true` costs a full workspace reload. Use it after a restore or build that was meant to repair unresolved references: that state lives in build output, which the file-set check cannot see, so an ordinary call can legitimately answer "unchanged". A restore that rewrote a project's `project.assets.json` after the session was loaded is the exception: an ordinary call reloads through MSBuild for it and names the project in `reason`. After a restore or build, an analysis that reported unresolved references retries once by itself; `force` is the deterministic way to demand it.
- A recalled session is rejected: `refresh_session` has no live workspace to refresh. Use `refresh_remembered` (or `analyze_solution`).

3.9 Staleness or incompleteness?

Two failure pictures look similar but need different reactions. The staleness check compares **source files**; incompleteness is about **project references** that the analysis could not resolve.

Picture	What is out of date	How it looks	What to do
Source staleness	the source files on disk differ from the files of the analysis	`not_found` on a type you just wrote (the file was never in the model); fan-in answers from a pre-edit graph	`refresh_session` - under `Reactive` / `Proactive` usually nothing, because the server repairs it first
Reference incompleteness	the model exists, but not all project references could be resolved	plausible-looking but **too short** fan-in answers; a leading incompleteness note on the affected tools; `incompleteProjects` in `get_diagnostics`; `verdict: "inconclusive"` when a strict majority of projects is incomplete	restore/build the solution, then `refresh_session(force: true)`
Analyzer drift	the server binary is old; the sources are untouched	nothing - the staleness check is correctly silent here	rebuild or reinstall the server; `server_info` reports the ANALYZER DRIFT line

Because the staleness check compares source files, it is silent by design when the binary is the stale part: an older analyzer keeps answering with its old facts and trips no source check. That is why `server_info` exists alongside it - the two signals see disjoint things.

3.10 The `dbPath` parameter

Many database-bound tools take an optional `dbPath` (for example `save_session`, `apply_solution_config`, `analyze_solution`, `list_insights`, `diff_review`). The resolution is:

explicit `dbPath` > the server's standard config DB > none.

When you omit it, the server resolves the standard config DB - the same database the GUI uses - if it resolved one at start; otherwise there is no default. An explicitly passed path always wins.

This matters most for the tools that **write**:

- Without `dbPath`, a writing call targets the live database of the GUI. `save_session` writes a manual snapshot into it; `apply_solution_config` additionally writes a git-tracked `.aicb.json` sidecar next to the solution; `diff_review` writes a regression log only when you pass `recordRegressions: true`.
- Every tool states its own write behavior in its description. Pass an explicit `dbPath` whenever you mean a different database.

For `analyze_solution`, a resolved database also supplies the analysis settings: its active namespace-exclusion list, its test-detection profile and (unless you pass an explicit `layerProfile`) its layer-mapping profile. The layer and test profiles travel with the session. Namespace exclusions travel only on the DB-free sidecar path; exclusions supplied by a database are applied to the initial analysis but are not retained for a later full refresh, a known limitation.

3.11 Timing constants via environment variables

Three wall-clock constants of the session machinery can be set per server process with environment variables. All three are in **milliseconds**.

Variable	Overrides	Built-in default
`AICB_MCP_DEBOUNCE_MS`	quiet period of the change watcher before it triggers a background re-analysis	`15000` (15 s)
`AICB_MCP_STALENESS_WINDOW_MS`	throttle window of the staleness check: how long a "current" answer is re-served before the disk is read again	`5000` (5 s)
`AICB_MCP_LOAD_WAIT_MS`	how long a caller waits behind a first-touch analysis of the same solution before it is told what is loading	`45000` (45 s)

Notes:

- A value that is unparseable, negative or **zero** falls back to the built-in default. Zero is refused rather than honored because both consumers would read it as "no wait at all": a zero debounce would fire a full re-analysis on every keystroke, and a zero throttle would put the directory walk on every single tool call.
- There is deliberately **no upper bound**. A nonsensical value costs the operator who typed it.
- `AICB_MCP_LOAD_WAIT_MS` is chosen against the **client's** own deadline, not against any load time: a message that arrives after the client has already given up is worthless. Only the MCP server applies this bound; the GUI and the CLI wait without a limit.
- The three variables belong to the server process. They are read when the value is first needed, so they must be set in the environment that starts `aicb mcp`.

4 Profiles, pools and facets

An MCP profile is one self-contained MCP working environment. It decides four things:

Axis	What it controls	Where you set it
Tools	which tools `tools/list` shows and `tools/call` accepts	`Facets` and `Tools` tabs of the profile editor
Instructions	the working style the server sends to the model on connect	`Style` switches on the `Facets` tab, `Instructions` tab
Rendering	the context template per kind of work, plus output notation, token budget and overshoot tolerance	template pickers on the `Facets` tab, `Output` tab
Session freshness	whether and when the server re-analyzes after your edits	`Session` tab

Exactly one profile is active, and only the active profile drives the server. The active profile is fixed for the lifetime of a server process - there is no runtime switch tool, and `list_mcp_profiles` is read-only. All profiles live in the config DB, by default `%APPDATA%\AIContextBuilder\user-data\aicb.acb`, which is the same database the GUI uses. A normal `aicb mcp` start resolves that database automatically, so the profile you activated in the GUI applies without any argument; pass `--db-path <db>` to work against a different database.

Four profiles ship with the product; everything else is a custom profile you create, duplicate or fine-tune in the GUI.

4.1 Pools: what the server registers and what it delivers

The server registers **82 tools** in **28 tool classes**. Registration is wholesale; what you actually see is the **active pool**, applied on every request by two filters that read the same name source. The active profile behind it is fixed at server start:

- **Listed** - `tools/list` is narrowed to the active pool.
- **Callable** - a call guard refuses every tool the listing hides, with this message:

```
Tool '<name>' is not in the active profile - call list_mcp_profiles to see the profiles and
their tools, then restart the server with 'aicb mcp --mcp-profile <id>' to run under a
different one.
```

`batch` and `measure` apply the same check per sub-query, so a sub-query naming a tool outside the active pool is refused item by item rather than silently executed.

The three pool states

`list_skills` groups every registered tool into exactly one of three lists. Under the shipped `Default` profile:

Group	Count	Meaning
`inPool.core`	24	the navigation core exposed by the shipped Default profile; core tools appear in no facet menu
`inPool.extras`	30	the rest of the active pool
`outOfPool`	28	registered, but the active pool does not expose it

The `outOfPool` list mixes three causes, and the way out differs per cause:

1. **18 tools the `Default` profile withholds.** They stay in the lean core and in facet menus, so switching to `Full Select` is enough.

2. **One bundle tool,** `compare_public_api`**.** It reaches no facet menu at all; it becomes available through the `api-surface` bundle (see "Opt-in bundles") or through `AICB_MCP_TOOLS`.
3. **Nine tools that no facet menu and no bundle names.** Only `AICB_MCP_TOOLS` exposes them - not even `batch` / `measure` can reach them: `import_constellation`, `inspect_session`, `list_constellations`, `list_remembered`, `list_run_templates`, `list_sessions`, `recall_codebase`, `refresh_remembered`, `remember_codebase`. `list_skills` names them in its own `uncatalogued` section and states why: they are session and memory plumbing plus aicb database browsing, not tools about the analyzed solution.

The navigation core

The core is the profile's starting surface before per-profile exclusions. Its 24 tools, grouped the way the `Tools` tab shows them:

Group	Tools
Analyze & render	`analyze_solution`, `export_markdown`, `refresh_session`, `get_diagnostics`
Navigate symbols	`find_symbol`, `symbol_signature`, `find_usages`, `find_implementations`, `find_overrides`, `get_type_hierarchy`, `impact_of_change`, `find_tests_for`, `call_graph`
Context bundles	`get_context`, `explain_symbol`, `pack_for_task`
Service & discovery	`server_info`, `usage_report`, `batch`, `measure`, `list_mcp_profiles`, `list_skills`, `docs`, `install_agent_hooks`

The core is never part of a facet menu - the two are added up, not merged, so a facet counter never counts a core tool twice. Every profile composes its tool set as: the core, plus the union of the enabled facets' menus, plus any opt-in bundle tools, minus the core tools you switched off on the `Tools` tab.

What the `Default` profile withholds

`Default` enables all nine facets but narrows every facet menu by one shared list of 18 tools:

Tool	Why it is withheld
`find_production_dead`	the least reliable of its family; `find_dead_code` stays available
`find_structural_twins`	reports similarity, not equivalence
`find_by_resource_leak`	low call volume, weak precision
`find_by_concurrency_risk`	its signal direction changed repeatedly
`check_doc_drift`, `check_pattern_drift`, `confidence_map`, `find_by_code_traits`, `find_by_complexity_and_coverage`, `find_by_returns_semantic`, `find_god_objects`, `lifecycle_of`, `list_insight_producers`, `trace_flow`, `type_dependency_path`	long tail: called too rarely to earn a schema in every prompt
`diff_public_contract`, `diff_review`, `semantic_diff`	two-session comparisons: each needs a snapshot booked by an earlier call, so none can be reached in a single exchange with a cold session

A withheld tool is not a denial. It stays registered, `list_skills` reports it under `outOfPool` as proof that it exists, and a profile switch to `Full Select` - or the `AICB_MCP_TOOLS` override - reaches it.

Reading the pool

`list_skills` is the authoritative map, not the tool list your agent harness displays. The harness shows only the active pool (54 tools under `Default`) while the registered set is 82; a reference derived from a harness list would lose 28 tools without a warning.

- `list_skills` (or `detail: "lean"`, the default) returns the tool index as bare names in their pool groups, plus the `core`, `facets`, `styles`, `bundles` and `uncatalogued` sections.
- `list_skills(detail: "full")` adds each tool's one-line description. An unknown `detail` value is rejected with the two valid spellings.

4.2 The four built-in profiles

Id	Name	Tools	Shape
`mcp-profile/default`	`Default`	54	all nine facets enabled; every facet menu narrowed by the shared 18-tool list; instructions-neutral
`mcp-profile/full`	`Full Select`	72	all nine facets enabled with their complete menus - the whole lean core
`mcp-profile/refactoring-focus`	`Refactoring Focus`	47	only the `Refactoring` facet, with a curated menu and its working style switched on
`mcp-profile/debugging-focus`	`Debugging Focus`	43	only the `Debugging` facet, with a curated menu and its working style switched on

All four render every facet through the profile-wide `ai-optimized` context template in `YAML` notation, and none of them carries free-text instructions of its own. The two focus profiles send their facet's guidance as standing instructions; `Default` and `Full Select` stay instructions-neutral, so they add nothing to what the server already sends.

`Default` is the everyday pool: 24 core tools plus 30 from the nine facet menus, so every kind of work has its tools available without configuration. It is what an unconfigured server delivers.

`Full Select` is the complete lean core: 24 core plus 48 facet-menu tools. It exists so that the narrowing of `Default` never costs you a tool - one profile switch brings back all 18 withheld ones. Choose it when you want flow tracing (`trace_flow`), the two-session comparisons (`diff_review`, `semantic_diff`, `diff_public_contract`), the confidence map (`confidence_map`), or the refactor-prioritization queries.

`Refactoring Focus` enables the `Refactoring` facet only, curated to 23 non-core tools (plus the 24 core = 47), and sends that facet's guidance as the working style. It keeps structure, diff, smell, refactor-list and dependency-injection tools - including `find_structural_twins`, `find_by_complexity_and_coverage`, `semantic_diff`, `resolve_injection`, `prepare_task`, `solution_metrics`, `find_god_objects`, `find_by_concurrency_risk`, `find_by_resource_leak`, `find_by_event_subscription` and the three markup tools. It drops:

- `list_insight_producers` - a static meta catalog, not a refactoring step
- `find_by_attribute` - generic attribute search
- `find_by_returns_semantic` - return-role classification
- `find_by_code_traits` - LINQ/reflection/throws audit
- `calls_external` - external-API audit

`Debugging Focus` enables the `Debugging` facet only, curated to 19 non-core tools (plus the core = 43), and sends that facet's guidance as the working style. It keeps the fault-tracing set, among them `trace_flow`, `type_dependency_path`, `lifecycle_of`, `verify_claim`, `review_context`, `find_by_code_traits`, `find_by_side_effects`, `find_by_concurrency_risk`, `find_by_resource_leak`, `describe_api_surface` and `find_by_event_subscription`. It drops the meta catalog, the generic name searches (`find_by_attribute`, `find_by_returns_semantic`), the audit and refactor lists (`find_dead_code`,

`find_production_dead`, `find_god_objects`, `find_by_complexity_and_coverage`, `solution_metrics`, `detect_circular_dependencies`, `calls_external`) and the three database/history-bound snapshot tools (`save_session`, `compare_with_previous`, `diff_public_contract`).

4.3 Selecting a profile

In the GUI

1. Open `MCP Profiles` in the settings tree.
2. Select the profile in the list on the left.
3. Press `Apply as Active` in the footer.

Exactly one profile is active; the `Active` pill in the list marks it. `Save` persists your edits, `Duplicate` (`Ctrl+D`) creates a custom copy as a starting point, `Delete` removes a custom profile permanently and hides a built-in (recoverable with `Restore Built-In`), and `Export as Built-In` shows the profile as a C# snippet (intended for the product's own development; it has no effect on your installation). `Ctrl+N` creates a new profile, `F5` reloads the list.

At server start (pin)

```
aicb mcp --mcp-profile mcp-profile/full
```

`--mcp-profile <id>` pins the active profile for this server process. The pin is process-local and writes nothing back: a second server process, and the GUI, keep their own active profile. Without the option, the persisted active profile applies. Get the ids from `list_mcp_profiles` or from the profile list in the GUI.

- An unknown id aborts the start, with the valid ids listed on stderr - a typo never silently serves a different profile.
- Hidden profiles are rejected as pin targets. They are not listed, so pinning one would be an invisible configuration.
- The pin needs a config DB. The standard database is used automatically unless you pass `--db-path`.

`list_mcp_profiles` still shows the whole catalog, with the pinned profile flagged `isActive` - not the one persisted in the database.

What a running server follows - and what needs a restart

Change	When it takes effect
A facet's template assignment	live, per call
Active profile changed in the GUI	at the next server start; until then `server_info` reports it as config drift and asks for a restart
Tool set and server instructions of the active profile	at server start - a running server keeps what it read at startup
`AICB_MCP_TOOLS`	at server start, resolved once

Note: the server does not send `tools/list_changed` notifications. A profile change therefore reaches a client's tool list only when the client re-reads it; the server marks its tool list cacheable for 15 minutes with a private cache scope, and that expiry is the only invalidation it offers.

4.4 Overriding the tool set

`AICB_MCP_TOOLS`

This environment variable replaces the profile composition wholesale for the process. It accepts:

Value	Result
`all`	every registered tool - 82
`lean`	the complete lean core - 72, the same set as `Full Select`
empty or unset	no environment override; `aicb.mcp.json`, then the active profile (Default: 54), decides
a comma-separated list of tool-class names	only the tools of those classes, e.g. `SymbolTools,MemoryTools`
`methods:<tool>,<tool>,...`	exactly the named tool functions

Class names are matched case-insensitively, and unknown names are ignored. If the list matches no class at all, the server falls back to the lean core rather than exposing nothing. Because the override is meant to be an operator decision, the set is resolved once at start and does not move when the active profile changes. A class list *replaces* the profile pool - it is not added to it - so name every class you want.

`aicb.mcp.json`

The server reads an optional, hand-written config file at `%APPDATA%\AIContextBuilder\aicb.mcp.json` (`<ApplicationData>/AIContextBuilder/aicb.mcp.json` on Linux and macOS). It is read-only for the server: you edit it by hand. All fields are optional.

Field	Meaning
`ToolSpec`	the same tool-set spec `AICB_MCP_TOOLS` accepts
`SessionCache.TtlMinutes`	session time-to-live in minutes; default 90
`SessionCache.MaxSessions`	upper bound on concurrently held sessions; default 8
`SessionCache.SweepMinutes`	interval of the background sweep in minutes; default 5

A missing file means "no override". A broken file is reported on stderr and the server starts on the defaults instead of failing. Property names are case-insensitive, and comments and trailing commas are tolerated.

Precedence

```
AICB_MCP_TOOLS  >  aicb.mcp.json  >  the active profile  >  the Default profile's pool
```

When neither the variable nor the file is set, the active profile decides. It is fixed at server start, so an edit in the GUI reaches the server only after a restart (`server_info` reports it as config drift until then). If reading the profile fails (a locked database, for example), the server falls back to the `Default` profile's pool instead of exposing nothing.

A profile begins with the navigation core, but every core tool except `server_info` can be switched off on the `Tools` tab. A heavily narrowed profile can therefore omit `list_mcp_profiles` and `list_skills`; `server_info` is the one core tool that cannot be switched off and carries a lock instead of a checkbox.

4.5 The nine facets

A facet is one kind of work. It is the axis that ties together a context template, a tool menu and a working style, and it is a call parameter, never server state: naming a facet in a call never changes `tools/list`. Every facet id is also the name of its render slot, and a call that names a facet receives that facet's guidance back whether or not the profile switched its style on.

Facet (`id`)	Title	Guidance	`Next:` hints
`general`	General	Balanced context work: navigate, read and verify with a broad, task-neutral selection.	`find_usages`, `get_context`, `get_diagnostics`

Facet (`id`)	Title	Guidance	`Next:` hints
`exploration`	Exploration	Understand an unfamiliar codebase before changing it: start from the architecture and the public surface, then drill into specific symbols.	`architecture_overview` , `explain_symbol` , `get_context`
`refactoring`	Refactoring	Prioritise structural clarity and dependency direction. Prefer minimal, behaviour-preserving edits; surface coupling/cohesion smells before restructuring.	`impact_of_change` , `evaluate_change_set` , `find_structural_twins`
`debugging`	Debugging	Trace control and data flow to the fault. Fix the root cause, not the symptom, with the smallest edit, and reason explicitly about reproduction steps.	`trace_flow` , `find_by_side_effects` , `get_diagnostics`
`review`	Review	Review a change end-to-end: blast radius, introduced findings and test coverage before it lands.	`review_context` , `diff_review` , `evaluate_change_set`
`testing`	Testing	Cover behaviour with focused, deterministic unit tests; find the gaps and pin invariants and edge cases. Keep production code and test code clearly separated.	`find_tests_for` , `coverage_gaps` , `prepare_task`
`documentation`	Documentation	Explain intent over mechanics. Keep summaries dense and accurate, and document the non-obvious 'why' against the real API surface.	`describe_api_surface` , `explain_symbol` , `confidence_map`
`architecture`	Architecture	Respect Clean/Onion layering: dependencies point inward, and Core/Contracts depend on nothing above them. Surface layer violations, cycles and coupling.	`architecture_overview` , `detect_circular_dependencies` , `resolve_injection`
`performance`	Performance	Identify hot paths and complexity hotspots. Avoid unnecessary allocations and LINQ in tight loops, and measure before optimising.	`symbol_metrics` , `find_by_complexity_and_coverage` , `calls_external`

Each facet has three independent effects:

1. **Context template** - through its render slot (see "Template resolution" below).
2. **Tool menu** - the facet's tools join the profile's tool set when the facet's `Tools` switch is on. Unfolding the facet row lets you fine-tune the individual functions.
3. **Guidance trailer** - when a call names the facet, the answer ends with a block in this form:

```
---
facet:<id> (<Title>): <Guidance>
Next: <tool>, <tool>, <tool>
```

If one of the suggested `Next:` tools is not exposed by the active profile, the block is followed by a note that repeats the unreachable names and names both remedies:

```
Not exposed by this server's active tool profile: <names> - a direct call is refused.
Enable this facet's tool menu in your MCP profile (list_mcp_profiles), or set AICB_MCP_TOOLS.
```

The facet block itself stays byte-identical whether or not the style is switched on. `general` is the one facet whose menu is a hand-picked list of 23 tools rather than a group of tool classes; measured, it contributes no tool the other eight facets do not also carry. It stays for two reasons: it is the menu a "General only" profile shows, and it is the default template slot - a call without a facet resolves to `general`.

The `facet` call parameter

Three tools accept an optional `facet` parameter: `prepare_task`, `pack_for_task` and `export_markdown`. It is stateless and costs nothing; the legacy spelling `slot` is accepted for it as well. For `prepare_task` and `export_markdown`, a call without a facet uses the profile-wide template and a call with a facet selects that facet's template and adds the guidance block. `pack_for_task` always renders its template-free lean bundle; its facet changes only the appended guidance block.

Template resolution

Each facet resolves its context template through this chain:

```
populated facet slot  >  the profile-wide template  >  the globally active context template
```

In all four built-in profiles the profile-wide template is `ai-optimized` and every other slot is unset, which the editor shows as `(use profile template)`. On the `general` row the unset sentinel reads `(use active context template)`. A template id that no longer resolves falls back gracefully to the `default` context template instead of failing the render.

Note: because every facet inherits the profile-wide template, structural facets such as `Architecture` and `Review` render without the full dependency graphs that a dedicated structural template would carry. If you want those graphs, assign `refactoring` to the `Architecture` and `Review` rows as a slot override.

4.6 Styles and bundles

Guidance styles

Two working styles cut across every kind of work. Each adds its text to the server instructions and never changes the tool set. They live on the `Instructions` tab.

Id	Title	Guidance
`async-correctness`	Async / concurrency correctness	Watch for async pitfalls: sync-over-async (.Result/.Wait()), missing CancellationToken, ConfigureAwait(false) in library code, and unobserved exceptions in async void.
`security`	Security review	Check for injection, unvalidated input, secret handling and unsafe deserialization. Prefer least privilege and fail-closed defaults.

The other six working styles are bound to a single facet and live on that facet's row as its `Style` switch - `Refactoring`, `Debugging`, `Architecture`, `Testing`, `Documentation` and `Performance`.

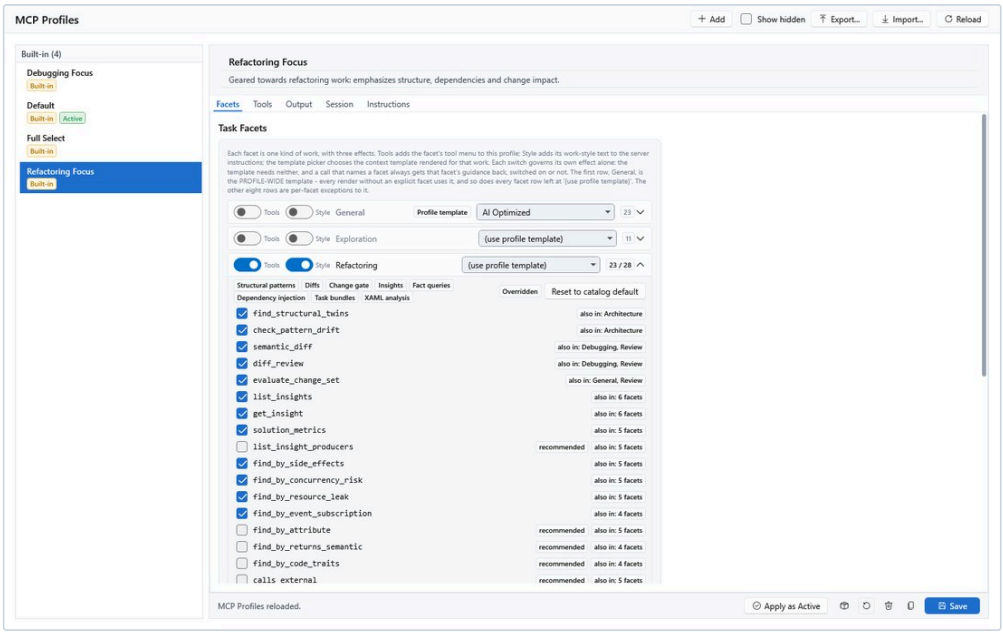
Opt-in bundles

A bundle groups tools by project type rather than by kind of work. On the `Tools` tab, `Opt-in Tool Bundles` lists the bundles that are not part of the standard pool. There is one: `Public API compatibility` (`api-surface`), whose tool is `compare_public_api`. Enabling it adds its tools on top of the facet menus, and it works on every composition path, including a profile with no facet selection at all. `AICB_MCP_TOOLS`, if set, overrides the whole composition.

The remaining bundles (`Quality & smells`, `Architecture & structure`, `Change impact & refactoring`, `Tests & coverage`, `UI / XAML markup`) are a legacy discovery grouping. Their tools are all reachable through the navigation core or a facet menu, and their ids have no effect as opt-in entries. `list_skills` therefore shows only the opt-in bundles in its `bundles` section.

4.7 The profile editor

The `MCP Profiles` panel is a master-detail editor: the profile list on the left, the selected profile on the right, with five tabs - `Facets`, `Tools`, `Output`, `Session` and `Instructions`. The header edits the profile's name and description. The footer carries the actions described under "In the GUI".



The MCP profile editor: profiles on the left, the task facets and their tool menus on the right

The `Facets` tab

`Task Facets` shows one row per facet, each with two switches, a template picker, a counter and an unfoldable function list:

- `Tools` - adds the facet's tool menu to this profile's `tools/list`. Unfold the row to check or uncheck individual functions.
- `Style` - adds the facet's working style to the server instructions. It is independent of the tools switch and never adds or removes a tool.
- **Template picker** - the context template for this facet. The `general` row sets the profile-wide template (marked with a `Profile template` pill) and reads `(use active context template)` when unset; the other eight rows are exceptions to it and read `(use profile template)` when unset.
- **Counter** - checked versus available tools on the facet's menu, for example `12 / 14`. While `Tools` is off, the counter names the menu size alone, because the facet contributes none of them. The navigation core is never part of this count.
- **Function list** - one checkbox per tool with its description, plus chips naming the tool classes that feed the menu and the other facets that also carry the tool. When your selection deviates from the catalog default, an `Overridden` pill appears next to `Reset to catalog default`.

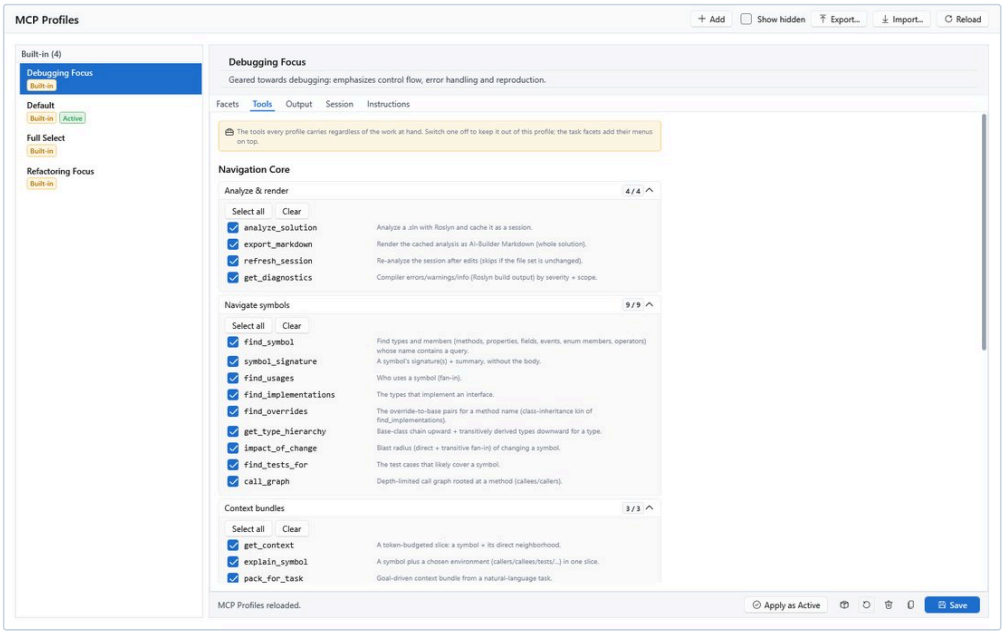
Each switch governs its own effect alone: a facet can send its working style with its tool menu switched off, and the template needs neither.

The `Tools` tab

`Navigation Core` lists the core tools in four groups (`Analyze & render`, `Navigate symbols`, `Context bundles`, `Service & discovery`), one checkbox per tool. Unchecking a tool removes it from this profile; `Select all` and `Clear` work per group. `server_info` has no checkbox - it is locked, so a mis-curated profile can always be diagnosed from the client side. The counter next to a group reads "exposed / total" and turns into a warning color while something in the group is switched off.

At five or fewer effective tools the panel shows a warning naming the actual count; the profile still composes, it is simply not worth connecting to.

`Opt-in Tool Bundles` follows, then `Effective tools/list`: the exact tool functions this profile exposes to the server on connect, as a read-only list. It is what the two sections above add up to.

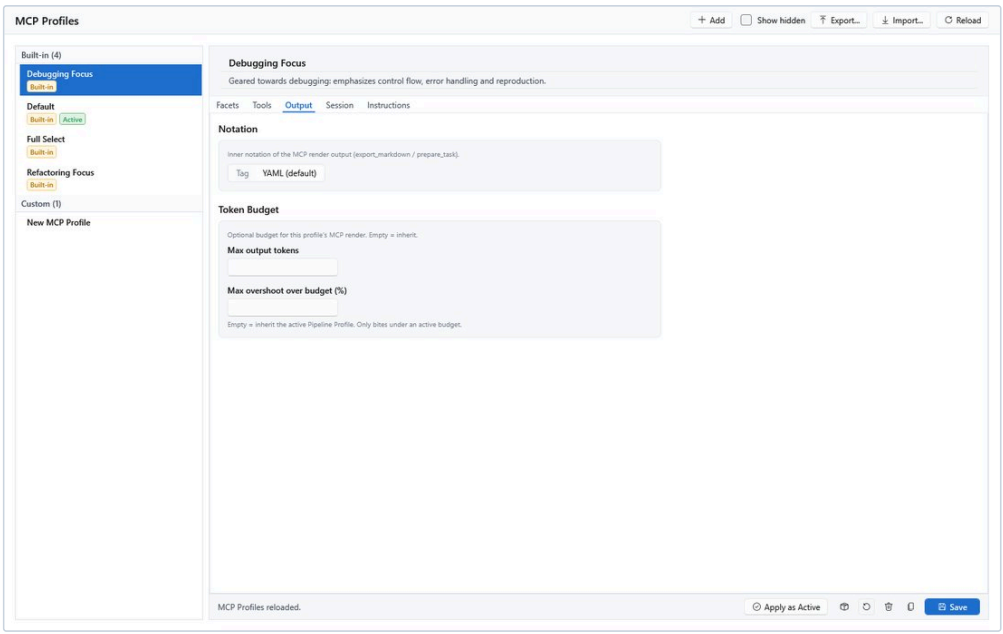


The Tools tab: the navigation core in four groups, the opt-in bundles and the effective tools/list

The `Output` tab

Setting	Values	Notes
`Notation`	`Tag` or `YAML (default)`	inner notation of the MCP render (`export_markdown` , `prepare_task`); an explicit `format=` argument on the call wins over the profile default; the `.md` file name is unchanged
`Max output tokens`	integer, empty = inherit	floored at 8000; when set it wins over the template budget; precedence: explicit call parameter > MCP profile > template > global
`Max overshoot over budget (%)`	integer 0-1000, empty = inherit	how far the template render may exceed the budget before over-budget selected types are dropped; `0` is a hard cap, a large value effectively never drops a selected type; empty inherits the active pipeline profile's percentage

The token budget and the overshoot tolerance affect the MCP render path only; GUI and CLI exports are unaffected. The token-budgeted single-symbol tools (`get_context` , `pack_for_task` , `explain_symbol`) keep their own transport cap and ignore the overshoot percentage.



The Output tab: notation, token budget and overshoot tolerance of the MCP render

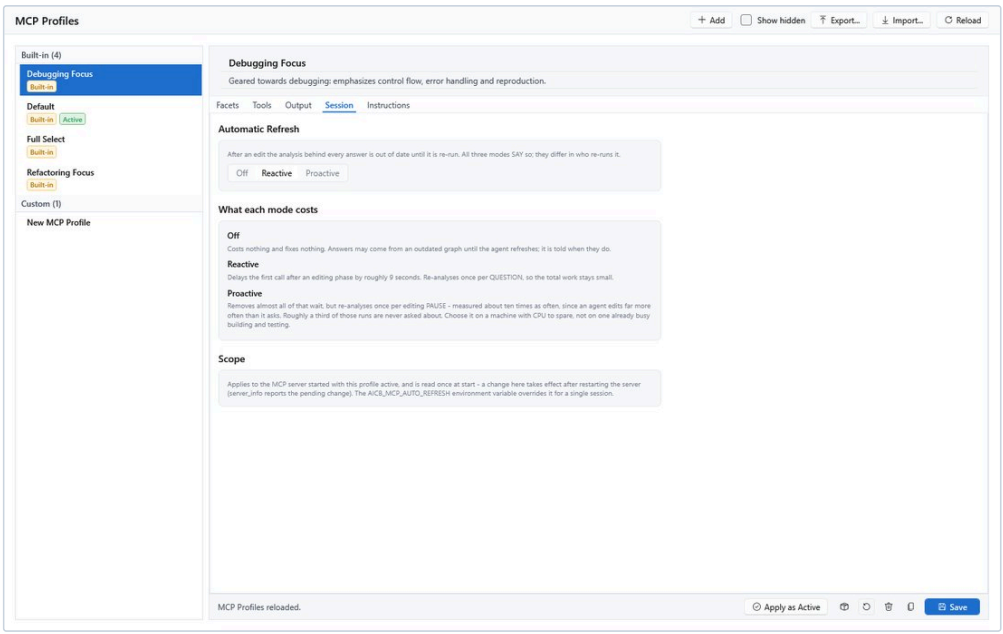
The `Session` tab

`Automatic Refresh` decides how the server keeps its analysis in step with the code on disk. All three modes disclose drift - a stale answer carries a note saying how old its graph is; the modes differ in who repairs it.

Mode	Behavior	Cost
`Off`	no automatic re-analysis; a drifted session is reported and left alone, and the agent repairs it by calling `refresh_session`	nothing - and it fixes nothing
`Reactive`	repair when asked: the drift is noticed as a tool is about to answer, the re-analysis runs first, and the answer comes from the current graph	the caller waits for the re-analysis
`Proactive`	repair ahead of the question: a file watcher starts the re-analysis once saving has gone quiet, so the next call usually finds a current graph	the work moves into the background and there is measurably more of it - roughly ten times as many runs, about a third of them never asked about; choose it on a machine with CPU to spare

All four built-in profiles, and every newly created custom profile, start on `Reactive`. The setting is read once at server start; a change takes effect after restarting the server, and `server_info` reports the pending change. The `AICB_MCP_AUTO_REFRESH` environment variable overrides the mode for a single process. It accepts `off` (also `0`, `false`, `no`), `reactive`, and `proactive` (also `1`, `on`, `true`, `yes`); an unrecognized value is ignored, so a typo cannot switch a mode on.

The `Scope` note on the tab summarizes this: the setting applies to the server started with this profile active, and the environment variable overrides it for one session. See "Sessions and staleness" for the full behavior of the freshness modes.



The Session tab: the three automatic-refresh modes with their costs

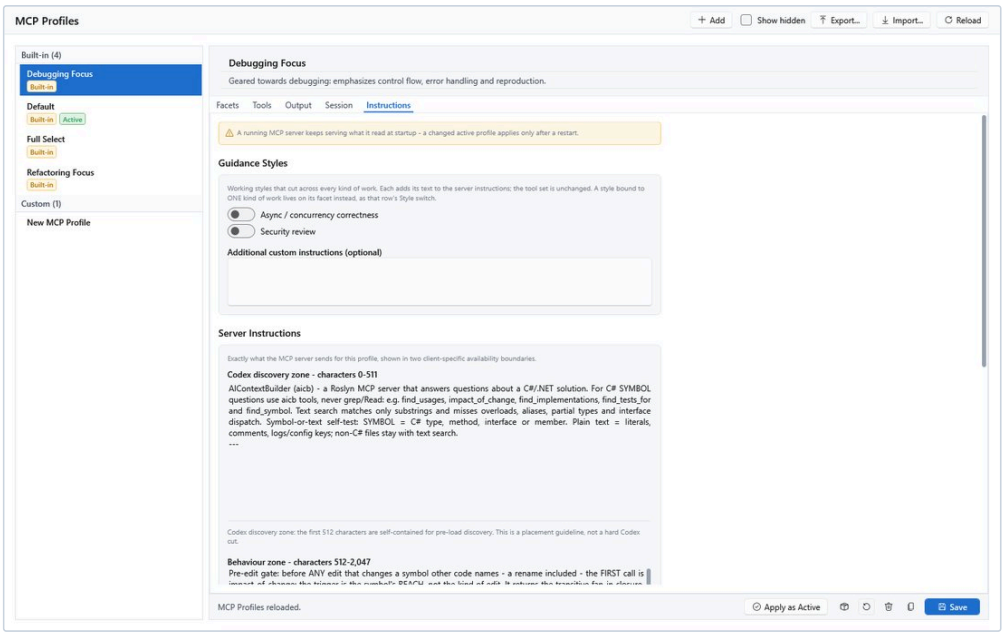
The `Instructions` tab

`Guidance Styles` carries the two cross-cutting styles as switches. `Additional custom instructions (optional)` is a free-text box appended after the facet guidance and the checked styles; write only what they do not already say, because a guidance text and a re-worded copy of it both reach the model - only exact duplicates are collapsed.

`Server Instructions` is a read-only preview of exactly what the server sends for this profile, split into three zones:

- `Codex discovery zone - characters 0-511` - self-contained identity and the C# symbol-versus-text switch for Codex before deferred tool loading.
- `Behaviour zone - characters 512-2,047` - the remaining behavior-changing rules and the profile working style inside Claude Code's measured prompt prefix.
- `Post-prefix reference zone - from character 2,048` - reference and discovery material outside that prefix; it is still transported by the server and available to Codex after tool loading.

A delivery cut mark appears when the text runs past the prefix, and the working style is shown on its own as well. The tab also counts how many of the tools the delivered text names this profile actually exposes, and names any that it withholds. None of this is a transport limit: switching on more guidance can push the later blocks past the client's prefix, and the order they compose in decides which ones arrive - facets first in catalog order, then the cross-cutting styles, then the free text.



Editing the server instructions of an MCP profile

Saving, restoring and the built-in freeze

`Save` (`Ctrl+S`) persists the editor's state to the database. Saving a built-in profile marks its row as overridden, and from that moment the built-in seed leaves the row alone.

Note: this has a lasting effect on `Default`. Its nine facet menus are stored as explicit lists derived from the 18-tool drop list, so the moment you press `Save` on `Default`, today's 54 tool names are frozen into that row and a tool cataloged into a facet by a later version no longer joins it. If you want the shipped default to keep growing with the product, do not save it. `Restore Built-In` clears the override and writes the current code default back; `Duplicate` leaves the built-in untouched and creates a custom copy you can edit freely.

4.8 Step-by-step

Switch to the complete tool set

1. Start the server with the pin: `aicb mcp --mcp-profile mcp-profile/full`, or open `MCP Profiles`, select `Full Select` and press `Apply as Active`.
2. Restart the client so it re-reads the tool list and the instructions.
3. Verify with `list_skills`: `inPool.core` plus `inPool.extras` should now total 72, and the 18 withheld tools should appear under `inPool.extras`.

Create a custom profile

1. Open `MCP Profiles` and press `Ctrl+N` (or select an existing profile and press `Ctrl+D` to start from a copy).
2. Give it a name and description.
3. On the `Facets` tab, switch on the kinds of work you want; unfold a facet to fine-tune its individual tools, and pick a context template per facet if the profile-wide one does not fit.
4. On the `Tools` tab, switch off any core tools you do not want, and enable the `Public API compatibility` bundle if you need `compare_public_api`.
5. On the `Output` tab, choose the notation and, if you need one, a token budget.
6. On the `Session` tab, choose the automatic refresh mode.
7. On the `Instructions` tab, switch on working styles and add free text if needed.
8. Press `Save`, then `Apply as Active`, then restart the server and reconnect the client.
9. Verify with `list_mcp_profiles` - your profile should be flagged `isActive` - and with `list_skills` for the effective pool.

5 Calling tools: conventions, batch and aicb call

Every aicb tool is a request over MCP, but there is more than one way to send it. This chapter covers the four ways a tool can be reached, the two meta-tools `batch` and `measure` that fan out over many read-only queries at once, the one-shot CLI invoker `aicb call`, the parameter aliases the server tolerates, which tools write, and the conventions that hold across the whole catalog.

5.1 The four call paths

A tool is reachable through up to four different doors. A tool that works through only one of them is a defect, so the doors are described here once for the whole catalog.

Path	What it is	What it reaches
1 · `tools/call`	The normal MCP path a connected client uses.	Exactly the tools of the **active tool pool**. A name outside the pool is refused even though it is registered.
2 · `batch` / `measure`	The read-only dispatch registry behind the two meta-tools.	The **48 dispatchable tools** (see below).
3 · `aicb call <tool>`	A one-shot invocation from the command line, **without a running server**.	**All 82 tools**, reading and writing alike - this verb has no curation.
4 · `tools/list`	The parameter surface your client is shown.	The same set as path 1: listing and callability are read from one and the same live source, so they cannot disagree.

How many tools each path reaches, depending on how the server was started:

Path	Shipped default	Under "Full Select"	With `AICB_MCP_TOOLS=all`
1 · `tools/call`	**54**	72	82
2 · `batch` / `measure`	**33** (48 registry entries minus the 15 outside the default pool)	48	48
3 · `aicb call`	**82** - no curation at all	82	82
4 · `tools/list`	**54**	72	82

Two properties of this table are easy to misread:

- **Path 3 records usage data only with `--db-path`.** `aicb call` does not use the `tools/call` pipeline; given `--db-path`, it writes the same row under the client name `aicb-call`, and without one it records nothing.
- **Path 2 records the parent call and a sub-query tally, but no row per sub-query.** The tally is described under "Counting sub-queries" below.

The refusal on path 1 reads:

```
Tool '<name>' is not in the active profile - call list_mcp_profiles to see the profiles and
their tools, then restart the server with 'aicb mcp --mcp-profile <id>' to run under a different one.
```

You cannot lock yourself out of the pool question: every profile includes the navigation core (the everyday tools plus `list_mcp_profiles`, `list_skills`, `docs`, `server_info` and `usage_report`), so `list_mcp_profiles` and `list_skills` are always callable and always tell you what the current pool contains. See "Profiles, pools and facets" for the profiles themselves and for `AICB_MCP_TOOLS`.

5.2 `batch`: many read-only queries in one round-trip

`batch` runs several **read-only** queries in one round-trip, sharing one analysis session. It is the right call when you already know you need several answers about the same subject - for example `find_usages` plus `get_type_hierarchy` plus `find_tests_for` for one symbol, `list_insights` followed by `get_insight`, or the markup fan-in trio next to a symbol query.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session shared by all sub-queries. Accepts the session ID from `analyze_solution` or an absolute `.sln` / `.slnx` / `.slnf` path to self-initialize once. Do **not** repeat it inside the sub-queries.
`queries`	array	yes	-	An array of `{ tool, args }` objects. `tool` is a read-only tool name; `args` is that tool's own argument object **without** `sessionId`. `args` may be omitted when the tool takes no arguments. Maximum **16**.

A query element is exactly `{"tool": "<name>", "args": { ... }}`. The `args` object may also be passed as a JSON **string**, which is unwrapped for you.

Rules and limits:

- **Maximum 16 sub-queries.** Exceeding the limit fails the **whole** batch with an actionable message rather than silently truncating - a silent cap would read as "all of them ran". Split a larger set into several calls. An empty `queries` array is rejected as well.
- **Sub-queries run sequentially**, in the order you list them.
- **A wrong `sessionId` fails the call once, up front** - not once per sub-query. This is also the only self-initialization point: a `.sln` path you pass is analyzed exactly once for all sub-queries.
- **Each sub-query is isolated.** One that fails reports `ok: false` plus its error while the others still answer normally. A cancelled call aborts as a whole.
- **The active pool governs the proxy path too.** A read-only tool that your active profile does not expose is refused per entry, with its own message - exactly as a direct call would be.

The response has the shape `{ results: [ {tool, ok, result | error} ], count, okCount }`:

Field	Meaning
`tool`	The tool name of the sub-query.
`ok`	`true` for an answer, `false` for a failure or an omission.
`result`	The answer. A JSON-returning tool's answer is nested as a JSON object (not an escaped string); a Markdown-returning tool's answer (for example `get_context` or `explain_symbol`) arrives as a string.
`error`	The failure message when `ok` is `false`.
`note`	An optional disclosure, omitted when there is nothing to disclose (see "Silently dropped arguments").

Response budget: 9,000 tokens. The answers are walked in order; an answer that is larger than the whole budget, or that would push the running total past it, is replaced by `ok: false` with its measured size and a pointer to call that tool directly. Only the **answer** is trimmed - the work was already done, so a dropped result costs latency but never correctness. In practice this only affects the slice tools, whose renders can be tens of thousands of tokens. `measure` has no such budget, because it has nothing large to return.

Identifying a sub-query in an error or omission. Because several sub-queries may name the same tool, failure and omission messages carry an echo of the sub-query's identifying arguments (`symbol=OrderService`, the first three scalar arguments) - inside the message text, not as a field of its own.

Why a sub-query was refused

`batch` and `measure` refuse tools for **four different kinds of reason**, and the refusal message names which one applies. Only the first is about writing.

#	Reason	Example names in the message
1	It **mutates session state or writes** - the read-only boundary.	`analyze_solution`, `refresh_session`, `apply_solution_config`, `save_session`, `remember_codebase`, `export_markdown`, `install_agent_hooks`
2	It needs a **second state** - another session or a stored snapshot, which one shared session cannot express.	`semantic_diff`, `diff_review`, `verify_claim`, `compare_with_previous`
3	**Recursion** - the meta-tools themselves.	`batch`, `measure`
4	It is **read-only but a poor batch member**: a heavyweight walk, renderer or multi-file payload whose answer would routinely exceed the response budget and be dropped after the work was paid for.	`evaluate_change_set`, `init_solution_config`, `prepare_task`, `review_context`

Note: the names in each bracket are **examples, not the whole group**. A tool's absence from those lists does not mean it writes anything - it only means the reason does not name it. The message itself says so. In particular, a number of non-dispatchable tools must be called directly (for example `docs`, `list_skills`, `server_info`, `list_mcp_profiles`, `usage_report`, `solution_config_status`, `check_solution_config_drift`). The last two are observational in purpose but can initialize a previously unknown solution row in the configuration database.

The final `Available:` line of the refusal names **every** dispatchable tool, regardless of your active profile. Since a sub-query outside the active profile is refused per item, take the tools you can actually dispatch from `list_skills` under `inPool`, not from that line.

A sub-query naming a tool the server does not register at all gets a different opening sentence - "is not a tool this server registers at all - check the spelling first" - so a typo and a real tool at the dispatch boundary are distinguishable. An empty tool name is reported as a malformed query object, not as a bad tool name.

The 48 dispatchable tools

These are the tools `batch` and `measure` can dispatch (alphabetical, as the refusal message prints them):

```
architecture_overview · assert_absence · call_graph · calls_external · check_doc_drift ·
check_pattern_drift · confidence_map · coverage_gaps · describe_api_surface ·
detect_circular_dependencies · explain_symbol · find_binding_usages · find_by_attribute ·
find_by_code_traits · find_by_complexity_and_coverage · find_by_concurrency_risk ·
find_by_event_subscription · find_by_resource_leak · find_by_returns_semantic ·
find_by_semantics · find_by_side_effects · find_dead_code · find_god_objects ·
find_implementations · find_overrides · find_production_dead · find_resource_usages ·
find_structural_twins · find_symbol · find_tests_for · find_unresolved_bindings ·
find_usages · get_context · get_diagnostics · get_insight · get_type_hierarchy ·
impact_of_change · instantiation_sites · lifecycle_of · list_insight_producers ·
list_insights · pack_for_task · resolve_injection · solution_metrics · symbol_metrics ·
symbol_signature · trace_flow · type_dependency_path
```

Fifteen of them (`check_doc_drift`, `check_pattern_drift`, `confidence_map`, `find_by_code_traits`, `find_by_complexity_and_coverage`, `find_by_concurrency_risk`, `find_by_resource_leak`, `find_by_returns_semantic`, `find_god_objects`, `find_production_dead`, `find_structural_twins`, `lifecycle_of`, `list_insight_producers`, `trace_flow`,

`type_dependency_path`) are outside the shipped default pool: they dispatch under "Full Select", `AICB_MCP_TOOLS=all`, or a profile that exposes them, and are refused per item otherwise. The other 34 registered tools are not dispatchable at all - call them directly.

Counting sub-queries

A `batch` (and `measure`) parent call records **what it dispatched**, so a tool reached only through `batch` no longer reads as never called:

- One increment per dispatched sub-query, grouped by tool name, plus a separate error counter. The user-visible form is `subQueries: [{ tool, calls, errors }]`.
- A sub-query counts as **failed** when it threw **or** was refused by the active pool. A refusal is a statement about your **profile**, not about the tool, so this error count is not comparable with a normal tool row's error rate.
- An unrecognized tool name collapses into the single literal `(unknown)`, so a typo cannot grow the key space - and the count itself is a signal about guessed tool names.
- **Not counted:** duration, result size, error text, and no row per sub-query. Correlation back to the caller is positional and by tool name only.
- When nothing was dispatched the tally is `null`, which applies to every ordinary (non-fan-out) call. Rows recorded before the tally existed simply omit it.
- `measure` **sub-queries are excluded from the aggregation.** `measure` rows are written, but every reader filters them out: it runs the tool fully, yet the caller consumes the **size** of the answer rather than the answer.

The tally is reported by `usage_report` under `subQueries` (uncapped, sorted by call count), and in the GUI's **MCP Usage** panel as the `Batch` column. Its tooltip is the exact reading instruction: *"How often the tool was DISPATCHED as a sub-query inside a batch. Those calls record no row of their own, so before this column a tool used only through batch read as never called. Read it as attempts, not runs: a sub-query the active profile refused, or one that failed on its arguments, is counted here too. measure is excluded - it runs the tool fully, but the caller consumes the SIZE of the answer rather than the answer."* The neighboring `Calls` column counts only direct calls.

5.3 `measure`: size an answer before you pull it

`measure` answers "how big **would** this answer be" - the exact token count, without returning the answer. Use it to decide before you pull: whether a call is worth making, which `budget` to pass to a slice tool (`get_context`, `explain_symbol`, `pack_for_task`), or whether to narrow `scope` first.

`measure` takes the **same parameters as** `batch` (`sessionId`, `queries`, maximum 16) and dispatches through the same read-only registry, so it accepts exactly the 48 tools above and refuses the same tools for the same four reasons.

The response is `{ measurements: [...], count, okCount, totalTokens }`. Per entry:

Field	Meaning
`tool`	The measured tool name.
`ok`	Whether the query succeeded.
`tokens`	The real tokenizer count (`cl100k_base`) of the exact text the tool would return.
`chars`	The character count of that text.
`returned` / `totalFound` / `truncated`	Passed through when the measured tool answers with a capped envelope, so you also see whether you would be seeing everything. Several axes sum, and `truncated` is `true` if any axis truncated.
`budgetNote`	For the Markdown slice tools, which have no envelope: the renderer's budget disclosure, passed through verbatim. If it says types were dropped, the answer you are pricing is already cut and a larger `budget` is what buys it back.

Field	Meaning
`error`	The failure message when the query failed.
`note`	An optional disclosure, as in `batch`.

`totalTokens` is the summed cost of all measured queries, so you can weigh several candidate calls without adding them up yourself.

Note: `measure` does **not** make a query cheaper to run. The server does the full work either way; only inspecting the answer becomes cheap - roughly 50 tokens instead of the answer. Measure to **decide**, then call once; do not measure every call by reflex.

Note: `measure` has exactly one steering deviation. For `export_markdown` - which cannot be dispatched and whose render can be tens of megabytes - the refusal does not say "call it directly" but gives the cheaper advice: give it an `outputPath`, which renders once, writes the file and returns the exact character count.

5.4 `aicb call`: one-shot invocation without a server

`aicb call <tool>` invokes one MCP tool from the command line without starting a stdio server. It is meant for validating any tool - including a brand-new or non-default-pool one - in a single process.

```
aicb call <tool> [--sln <path>] [--arg name=value]... [--db-path <path>]
```

Option	Meaning
`<tool>`	The MCP tool name to invoke, for example `find_usages`, `find_production_dead`, `server_info`.
`--sln <path>`	Absolute `.sln` path, bound to the tool's `sessionId` or `solutionPath` argument. Self-initializes and analyzes it once. Omit it for a tool that needs no session, such as `server_info`.
`--arg name=value`	A tool argument. Repeat the option for each argument, for example `--arg symbol=OrderService --arg includeTests=true`.
`--db-path <path>`	Optional config DB, the same one the GUI uses. DB-defaulting tools and self-init then apply its per-solution configuration. Without it the tool runs DB-free.

Examples:

```
aicb call server_info
aicb call find_usages --sln C:/repo/App.sln --arg symbol=OrderService
aicb call find_implementations --sln C:/repo/App.sln --arg symbol=IInsightsService
aicb call batch --sln C:/repo/App.sln --arg queries='[{"tool":"find_usages","args":{"symbol":"OrderService"}},
{"tool":"get_type_hierarchy","args":{"typeName":"OrderService"}}]'
```

How an argument is resolved. In this order: an explicit `--arg name=value`; the accepted alias spelling of that tool (see below; the real name always wins); for `sessionId` and `solutionPath` only, the value of `--sln`; the parameter's default; otherwise an error of the form `missing required argument '<name>': pass --arg <name>=<value>` (plus `(or --sln <path>)` for a session parameter). A `--arg` token without `=` is a user error.

How a value is converted. `string`, `bool`, `int`, `long`, `double`, `enum` (case-insensitive) and `string[]` (a JSON array or a comma-separated list) are read directly. Any other type is deserialized from JSON with the same options as `tools/call`, which is how a complex argument such as `batch`'s `queries` can be passed as `--arg queries='[...]'`. Numbers are read with the invariant culture: write `0.6`, not `0,6`. A decimal comma is rejected with a hint naming the cause, because on a German, French or Spanish desktop `0,6` looks perfectly well formed.

Output and exit codes. A string result is printed unchanged; anything else is serialized as JSON. The result goes to **stdout**, so it can be piped or inspected. Diagnostics and warnings go to **stderr** - including the progress of `analyze_solution`, which has no client to notify here and therefore reports on stderr. Where a note would follow a JSON array - for example the built-in-profile note of `list_insights` without `--db-path` - it is written to stderr as well, so stdout stays one JSON document. Exit codes: `0` success, `1` user error (unknown tool, missing or invalid argument, the tool's own actionable error), `2` unexpected failure, `3` cancelled.

An unknown tool answers `unknown tool 'X'. Available tools: ...` and lists the whole assembly - which is correct here, because this verb has no curation.

Note: `aicb call` reaches **every** tool, including the writing ones, and nothing stops you from invoking `save_session` or `apply_solution_config` through it. Two practical limits apply:

- Tools that need a live solution require a successful MSBuild registration; tools that analyze a solution fail if that is unavailable, while session-less tools still run.
- Tools with a DB default need `--db-path`. Without it the telemetry sink is a null sink, so `usage_report` answers `available=false`. Missing optional paths produce `dbPath is required` in `save_session`, `compare_with_previous`, `diff_public_contract`, `apply_solution_config` and `solution_config_status`. The required `dbPath` argument of `remember_codebase`, `refresh_remembered`, `import_constellation`, `list_remembered` and `recall_codebase` is rejected earlier by the command binder as `missing required argument 'dbPath'`.

5.5 Parameter aliases

The server accepts a second spelling for some parameters. `tools/list` keeps advertising the **real** name only: an alias is a tolerance, not a contract change. `usage_report` counts how often an alias was applied per tool (`aliasApplied`), so the tolerance stays measurable.

Tool	Real parameter	Also accepted
`find_implementations`	`interfaceName`	`symbol`, `interface`
`instantiation_sites`	`typeName`	`symbol`
`get_type_hierarchy`	`typeName`	`symbol`
`lifecycle_of`	`typeName`	`symbol`
`find_symbol`	`query`	`symbol`, `name`
`resolve_injection`	`interface`	`symbol`, `interfaceName`
`find_overrides`	`methodName`	`symbol`
`call_graph`	`method`	`symbol`
`find_by_attribute`	`attribute`	`attributeName`
`calls_external`	`pattern`	`apiName`
`export_markdown`, `prepare_task`, `pack_for_task`	`facet`	`slot`

The real name always wins: if you pass both, the canonical argument binds and the alias is the one that is dropped. Aliases are accepted on all three binding paths - `tools/call`, a `batch` / `measure` sub-query, and `aicb call`.

Note: `analyze_solution(path)` is deliberately **not** aliased. It takes three path-shaped parameters (`solutionPath`, `layerProfile`, `dbPath`), so a bare `path` does not identify one, and a silently wrong bind would be worse than the honest failure it replaces.

5.6 Silently dropped arguments

The MCP SDK binds the arguments it recognizes and drops the rest without a word. aicb discloses this instead of refusing: a call that named an argument the tool does not declare succeeds, but the answer carries a notice naming the dropped arguments - up to eight of them, then `(+N more)` - and states that the answer was computed as if they had not been sent, followed by the tool's real parameter list. The notice is an HTML comment beginning `<!-- ignored-arguments:` so it is machine-readable.

In `batch` and `measure` this disclosure cannot ride on the call envelope, because the entry point only sees `{sessionId, queries}`. It therefore travels per result entry in the `note` field, and aliases are resolved first - a call that used an accepted alias is not reported as ignored.

5.7 Which tools write

Twelve of the 82 tools change something; **70 are read-only**. Of the twelve, two only change the in-memory session state, and ten can write to your file system or a database. No writing tool modifies source code that Roslyn reads.

Write target	Tools	Notes
Session state only (in memory)	`analyze_solution`, `refresh_session`	Nothing on disk changes.
File system, path named by the caller	`export_markdown`	Writes the rendered Markdown only when `outputPath` is set; without it the render is returned instead.
File system, inside the project directory	`apply_solution_config`, `install_agent_hooks`	The only two tools that write into your repository: `apply_solution_config` writes the git-tracked `<SolutionName>.aicb.json` sidecar next to the `.sln` (commit it so the configuration travels with the repo); `install_agent_hooks` writes the guard script plus the harness wiring (`.claude/settings.json`, `.codex/hooks.json` or `opencode.json`) - and only inside the directory of the analyzed solution.
aicb database	`save_session`, `remember_codebase`, `refresh_remembered`, `import_constellation`, `apply_solution_config`, `solution_config_status`, `check_solution_config_drift`, `diff_review` (with `recordRegressions: true`)	The two status/drift tools ensure a solution record and can insert it when the solution is not known yet; otherwise they only read. See the `dbPath` note below.

Every writing tool discloses its write in its own `tools/list` description. `diff_review` writes only on explicit opt-in and says so: *"Read-only unless you pass recordRegressions."*

Note: for `save_session`, `apply_solution_config` and `diff_review(recordRegressions: true)`, an omitted `dbPath` resolves to the server's **standard config DB - the same database the GUI uses**. The caller then writes into the application's live data without the call looking like it. Pass an explicit `dbPath` if that is not what you want. The memory tools are the opposite: `remember_codebase`, `refresh_remembered` and `import_constellation` require an explicit `dbPath` and never fall back to the standard config DB. When no DB is configured at all, `diff_review(recordRegressions: true)` writes nothing, still succeeds, and reports the reason in `regressionLog.skippedReason`.

5.8 Recurring conventions across the catalog

So the individual tool reference does not have to repeat them, the cross-cutting rules are collected here.

`sessionId` and self-initialization

Most tools are session-bound: their first parameter is `sessionId`, and it accepts either the session ID returned by `analyze_solution` (or by `recall_codebase`) **or** an absolute `.sln` / `.slnx` / `.slnf` path, which self-initializes the analysis once. In `batch` and `measure` the session is passed once at the top level and shared by all sub-queries. Session-less tools include `server_info`, `usage_report`, `list_skills`, `list_mcp_profiles` and `docs`. See "Sessions and staleness" for the session lifecycle.

`scope`

Almost every fact query takes `scope`, default `"solution"`, or a **namespace prefix** such as `"MyApp.Core"` to narrow it. Two exceptions:

- `find_resource_usages` and `find_binding_usages` take a **path substring** of the markup files instead.
- `get_diagnostics` treats `scope` explicitly as a **post-filter, not a narrowing of the work**: all projects are compiled either way, so `scope` saves response tokens, not time.

`includeTests`

The default is `false` almost everywhere - the tools report production code first. Most tools say when the filter actually removed something by returning a dedicated count:

Field	Reported by
`testMatchesFiltered`	the `find_by_*` queries and `calls_external`
`testMethodsFiltered`	`find_dead_code`, `symbol_metrics`
`testTypesFiltered`	`find_dead_code`
`testImplementationsFiltered`	`find_implementations`
`testRegistrationsFiltered`	`resolve_injection`
`testUsersFiltered`	`find_usages` (only on `includeTests: false`)

`detect_circular_dependencies` is the exception: it excludes test projects when `includeTests` is false but has no filtered-count field, so namespaces or cycles removed with the test graph are undisclosed. Another exception with a different meaning is `prepare_task(includeTests)`, whose default is `true` and which controls whether the **covering tests are bundled into the answer** - not which code is searched. `find_usages(includeTests)` defaults to `true` as well; `false` leaves the users declared in test projects out before its 100-item cap and counts them in `testUsersFiltered`.

The capped envelope

Many list answers come as `{ items, count, totalFound, truncated }`. `count` is the length of the list you received, `totalFound` the true total, and `truncated` says whether anything was cut. Typical caps:

Cap	Where
200	most fact queries, semantics and markup queries, `describe_api_surface`
100	`find_symbol`, `find_usages`, `find_implementations`, `find_overrides`, `get_type_hierarchy` derived types, diff lists, `find_structural_twins`, dependency cycles
50	`symbol_signature`, `find_tests_for`
20	`impact_of_change` direct impact
500 edges	`call_graph`

`nearest`

When a name resolves to nothing, many tools add a `nearest` suggestion - the closest declared name, or a declared name the query only begins - instead of returning a bare empty list. That distinguishes "you misspelled it" from "this symbol really has no matches". It is present on `find_symbol`, `symbol_signature`, `find_usages`, `impact_of_change`, `symbol_metrics`, `instantiation_sites`, `get_type_hierarchy` and `resolve_injection`, and as a did-you-mean inside the `hint` of `find_implementations`.

`note`

A leading `note` field qualifies an answer that would otherwise be misread - above all the zero. The recurring pattern: a zero gets a denominator (how much was examined at all) and an explanation of which constructs the detector structurally cannot see.

`incompleteProjects`

If the analysis run could not resolve a project's references, a `note` leads **every** answer of the fan-in family, non-empty ones included: such a run is short by ordinary edges. The cross-check is `get_diagnostics`, which reports the affected projects as `incompleteProjects` together with their suppressed-diagnostic counts and their dependent projects.

A narrower qualifier of the same kind is `generatedCodeNote`. It appears when a source generator or the WPF markup compiler did not run in the project that declares the answer's subject, or in a project depending on one, and it names the generators that did not run together with the repair `get_diagnostics` gives for its `generatedCodeGaps`. It rides on populated answers as well. It is carried by `find_usages`, `impact_of_change` (ahead of `risk`), `instantiation_sites`, `find_tests_for`, `find_implementations` (on an empty answer also when generated code is missing below the interface's project) and `get_type_hierarchy`. `find_dead_code` marks each candidate in such a project with `missingGeneratedCode: true` and counts them in its `generatedCodeNote` - treat them as unjudged, not as dead. `find_symbol` and `symbol_signature` add the sentence to `generatedSourcesNote` on a zero hit.

Recall safety

Many tools state in their own description that they "work on recalled sessions" - they read persisted Roslyn facts and therefore answer on a session recalled from a database snapshot. The **live-only** tools are marked separately:

`get_diagnostics`, `check_doc_drift`, `init_solution_config`, `check_solution_config_drift`, `resolve_injection`, `evaluate_change_set`, `diff_review` (both sessions must be live), and `find_unresolved_bindings`, which runs against the live analysis and reports no findings on a recalled snapshot - and says so when the session looks recalled. `refresh_session` likewise rejects a recalled session; use `refresh_remembered` or `analyze_solution` to get a live one.

The run facts behind several notes are stored with the snapshot, so a recalled session carries the same notes as the live one: which `.cshtml` views the Razor generator did not compile (`VIEWS NOT READ`), which generated C# the index skipped, which Razor components exist only in markup, which projects were compiled without a generator's code (`generatedSourcesNote`, `generatedCodeNote`, `missingGeneratedCode`), and the creations, external calls and side effects of top-level statements. A session recalled from a snapshot stored before a fact was recorded cannot know it, and says so instead of answering as if nothing was missing. Each such note names `refresh_remembered` as the remedy:

Note opens with	Appears on
`VIEWS NOT KNOWN`	where `VIEWS NOT READ` would: a zero fan-in of `find_usages` or `impact_of_change`, and a non-empty candidate list of `find_dead_code` or `find_production_dead`
`GENERATED CODE NOT KNOWN`	a zero-hit `find_symbol` or `symbol_signature`; a `not_found` answer of `find_usages`, `impact_of_change` or `get_type_hierarchy`, a plain typo included; the fan-in, construction, declaration and tests answers about a declared subject; an empty `find_implementations` for an undeclared name; a non-empty `find_dead_code` list
`TOP-LEVEL CREATIONS NOT KNOWN`	`instantiation_sites`, `lifecycle_of` and the `find_usages` constructor form `Type.Type`, also on a populated answer, where the scope admits top-level sites

Note opens with	Appears on
`TOP-LEVEL EFFECTS NOT KNOWN`	an unscoped `calls_external` query and an unscoped `find_by_side_effects` query other than `purity='pure'`, also on a populated answer

A snapshot written by an earlier version lacks all of these facts, so on such a recalled session many answers carry one of the notes, even for a solution that has no views or top-level statements. `refresh_remembered` clears them.

6 Tool reference: orientation, sessions and symbols

This chapter documents the MCP tools that orient you on a solution, manage the analysis session, and find individual symbols. It covers three groups:

- **Orientation, operating manual and server introspection** - `docs`, `list_skills`, `server_info`, `list_mcp_profiles`, `usage_report`, `architecture_overview`, `describe_api_surface`
- **Sessions and analysis** - `analyze_solution`, `refresh_session`, `get_diagnostics`, `inspect_session`, `list_sessions`
- **Finding symbols, signatures and single-symbol context** - `find_symbol`, `symbol_signature`, `get_context`, `explain_symbol`

The fan-in and impact tools (`find_usages`, `impact_of_change`, ...), the quality, testing and markup tools, and the multi-query meta-tools `batch` and `measure` are documented in the other tool-reference chapters.

6.1 About this chapter

Session argument. Most tools here take a `sessionId`. You can pass either the id returned by `analyze_solution` or the absolute path to a solution file (`.sln`, `.slnx` or `.slnf`). When you pass a path, the server analyzes the solution on first use and reuses it afterwards, so `analyze_solution` is optional. If the value is neither a known session id nor an existing solution file, the call fails with a message that says which of the two it was read as and what to do next: a solution path that does not exist is reported as such (and explicitly *not* as an expired session), a solution path on a host that cannot self-initialize points you to `analyze_solution`, and anything else is reported as an unknown or expired session id.

Tool pool. The server exposes a curated subset of its tools rather than all of them, because a long tool list measurably degrades an agent's tool choice. `core` tools are unioned into every profile; `extras` are the rest of the default pool; a tool marked "not in the default pool" exists but the active tool set does not expose it. The environment variable `AICB_MCP_TOOLS` (the values `lean` or `all`, or a comma-separated list of tool-class names) is an operator override; the configured profiles are visible through `list_mcp_profiles`, and the active one is pinned at server start with `aicb mcp --mcp-profile <id>`.

Dispatch through `batch` and `measure`. Tools marked as dispatchable below can be run as a sub-query of the `batch` and `measure` meta-tools; the others must be called directly.

Capped lists. Many tools return `{ items, count, totalFound, truncated }`: `count` is the length of the delivered list, `totalFound` the true total. The caps in this chapter are 100 (`find_symbol`), 50 (`symbol_signature`), 200 (`get_diagnostics`, `describe_api_surface`) and 80 entries per macro section in `architecture_overview`, whose layer map lists up to 2000 types and only per-layer counts above that.

`nearest`. When a name resolves to nothing, several tools return the closest declared name instead of a bare empty list, so a typo is distinguishable from a symbol that genuinely has no matches.

Leading `note`. A leading `note` field qualifies an answer that would otherwise be misread - above all a zero: the zero gets a denominator (how much was examined at all) and an explanation of which constructs the detector structurally cannot see.

Writes. All tools in this chapter are read-only, with two exceptions: `analyze_solution` and `refresh_session` change the in-memory session state. Neither writes files or a database.

Tool	Purpose	Pool	`batch` / `measure`
`docs`	aicb's own operating manual	core	no
`list_skills`	capability map of the server	core	no
`server_info`	version, build commit, drift checks	core	no
`list_mcp_profiles`	the MCP profiles in the config DB	core	no
`usage_report`	server-side tool-call telemetry	core	no

Tool	Purpose	Pool	`batch` / `measure`
`architecture_overview`	bounded whole-solution orientation	extras	yes
`describe_api_surface`	public API surface of a solution or namespace	extras	yes
`analyze_solution`	analyze and cache a solution	core	no
`refresh_session`	re-analyze the session's solution after edits	core	no
`get_diagnostics`	compiler errors and warnings without a build	core	yes
`inspect_session`	metadata about one cached session	not in the default pool	no
`list_sessions`	all currently cached sessions	not in the default pool	no
`find_symbol`	find a symbol by name	core	yes
`symbol_signature`	signature without the body	core	yes
`get_context`	token-budgeted single-symbol slice	core	yes
`explain_symbol`	symbol plus a chosen environment	core	yes

6.2 Orientation, operating manual and server introspection

`docs`

Purpose. The aicb operating manual - how to *use* the server, as opposed to what it can tell you about your code. The content is served by the same server that needs explaining. No session and no solution are needed.

When to use. Call it when you are new to aicb, when you want to know how a feature is meant to be driven, or when you need the vocabulary used in other answers.

Parameters

Parameter	Type	Required	Default	Meaning
`topics`	`string[]`	no	`null`	Page or route ids. Omit for the directory. A route id expands to its pages in reading order; repeats collapse, so asking for a route and one of its pages yields one copy. Unknown ids are named back to you rather than silently dropped.

Pages and routes

Id	Content
`overview`	What aicb is (and is not): the two products, the four things it deliberately does not do, and the three ways to run it.
`install`	Installing aicb and wiring it into a client: the dotnet tool, `aicb init`, and what it writes - the client's `.mcp.json` entry, the agent skill a tool install cannot deliver, and the blocking symbol guard.
`first-context`	From a `.sln` to your first context: self-init, then choosing between an overview, a symbol slice and the full export - and why the full export is rarely right.
`navigation`	Asking questions about the code: the question-to-tool table, the blast-radius call to make before editing shared code, and where plain text search is still right.
`solution-config`	Configuring a solution: layers, exclusions and test detection, the three setup calls, and why to commit the `.aicb.json` sidecar.
`glossary`	Glossary: session, facet, MCP profile, snapshot, insight, AI-Builder Markdown, tool pool.
`init` (route)	Every page, in reading order - the whole first run.

Response. Markdown. Without `topics` you get the directory: one line per page and per route, each with an estimated size in tokens so you can budget before pulling. The estimate is a real tokenizer count, not a characters-divided-by-four rule of thumb; when no counter is available the size column is simply left out. If a requested id matches nothing, the answer names it and appends the directory.

Notes and limits

- If a rendered page suggests a tool that the active tool profile does not expose, a trailing note names those tools and both ways to widen the pool: an MCP profile (see `list_mcp_profiles`) or the `AICB_MCP_TOOLS` environment variable. The rest of the page applies unchanged.
- The same content is available as MCP resources - `acb://docs` for the directory, `acb://docs/{topic}` for a page or route - for clients that prefer attaching context over calling a tool.
- Not dispatchable through `batch` / `measure`.

`list_skills`

Purpose. The server's capability map: a tool index naming *every* tool this server has, each exactly once, grouped by pool state - plus the sections that group tools by name: the always-on navigation core, the task facets (id, guidance, template slot and the facet's tool menu), the cross-cutting guidance styles, and the legacy functional bundles.

When to use. To discover specialized long-tail tools beyond the navigation core, and - the one question no other tool answers - to check whether a tool you want is merely outside the active pool rather than nonexistent.

Parameters

Parameter	Type	Required	Default	Meaning
`detail`	`string`	no	`null` (equals `"lean"`)	`"lean"` - bare names in their pool groups; `"full"` - additionally each tool's one-line description. Case-insensitive.

Response. JSON with these fields:

Field	Content
`about`	How to read the index (including the reachability rule below).
`tools.inPool.core`	The tools every profile exposes, listed in reach-for-them order, not alphabetically.
`tools.inPool.extras`	The rest of what the active tool set exposes right now, alphabetical.
`tools.outOfPool`	Tools that exist but are not exposed by the active tool set.
`core`	The navigation core section (about text + tool names).
`facets`	Each facet with `id`, `title`, `guidance`, `templateSlot` and its tool menu.
`styles`	The cross-cutting guidance styles with `id`, `title`, `guidance`.
`bundles`	The opt-in bundles with `id`, `title`, `about` and their tools.
`uncatalogued`	Registered tools that no facet menu and no bundle names (about text + names).

In the lean projection each index entry is a bare name; with `detail: "full"` each entry is an object with `name` and `description`.

Notes and limits

- A name in `outOfPool` is proof the tool is real, not a denial. It becomes reachable through an MCP profile whose facet menu or bundle names it, or through the `AICB_MCP_TOOLS` override - except an uncatalogued tool, which no menu names, so only the override applies.

- An unknown `detail` value is rejected with the two valid values, rather than silently answered as the default.
- No session needed. Not dispatchable through `batch` / `measure`.

`server_info`

Purpose. Name, version and build commit of the server - a quick reachability check - plus the config DB's schema version. The commit answers the question a version number cannot: *is the binary I am talking to built from the code I just landed?*

When to use. To confirm the server is reachable and current, and to diagnose a stale server after a rebuild, an update or a profile change.

Parameters. None.

Response. Plain text, built from up to five parts:

1. The identity line: `aicb MCP server (AIContextBuilder) v<version> [(commit <sha>)] - Roslyn-based .NET context generator.` followed by the license notice (`(c) Gregor Dadera - free for private, hobby and educational use and qualifying organizations; see LICENSE.txt.`). A development build can include the full SHA; public release builds deliberately omit it.
2. `Config DB schema: user_version=<n> (<path>).` - a read-only read of the resolved config DB (the same one the DB-bound tools default to), so a post-migration check can read the schema version directly. Omitted when the server is DB-free or the DB is unreadable.
3. **Analyzer drift** - this binary's build commit against the HEAD of the repository holding the analyzed solution. It reports in-sync and ahead too, not only drift. Silent without an analyzed session, and on a repository that does not know this commit (any foreign solution). When git does not answer in time, it says `drift UNKNOWN` instead of staying silent.
4. **Version drift** - this binary against the config DB. It distinguishes a DB newer than this binary from a DB with pending migrations. For tool-pool drift it also distinguishes a profile produced by a newer build (rebuild/reinstall the standalone tool) from retired tool names still stored in the profile (open and save the profile in the GUI). The check is at tool-*name* level; a new parameter on an existing tool is not detected.
5. **Config drift** - the active MCP profile changed since this server started (a GUI edit in the "MCP Profiles" panel, or another process writing the config DB). Remedy: restart or reconnect to load the current skill pool (tools and instructions).

Notes and limits

- When a development build reports a commit, compare it against `git rev-parse HEAD`. A version number can be identical across *different* builds; public release builds provide no commit-level comparison.
- Parts 3-5 are omitted when there is nothing to report; the config-DB line is omitted on a DB-free server.
- Not dispatchable through `batch` / `measure`.

`list_mcp_profiles`

Purpose. List the MCP profiles in the server's config DB.

When to use. To see which profiles exist, which one is active, and how each one narrows the tool set and shapes the rendered output - for example before deciding which profile to pin at server start.

Parameters. None.

Requirements. The server must be started with `--db-path`. Otherwise the call fails with a clean message: `MCP profiles require a config DB. Start the server with: aicb mcp --db-path <db>.`

Response. JSON, one entry per profile:

Field	Content
`id` / `name`	The profile's identifier and display name.

Field	Content
`configuredSlots` / `availableSlots`	The configured template slots and the slots available to the profile. `General` appears in the configured set exactly when a profile-wide template is set.
`templateId`	The profile-wide template id; `null` means none.
`hasSkill`	Whether the profile carries a skill.
`toolSelection`	The effective tool set: the class-level CSV, or a `methods:`-prefixed function-name list when the profile curates individual tool functions; `null` means the lean core.
`isActive`	Whether this is the active profile.
render config	`tokenBudget`, `outputFormat` and `maxOvershootPercent`. A `null` `tokenBudget` or `maxOvershootPercent` means inherit: the template, then the active pipeline profile decides.

Notes and limits

- There is no runtime switch. Pin the profile at server start:

```
text aicb mcp --mcp-profile <id>
```

- Not dispatchable through `batch` / `measure`.

`usage_report`

Purpose. The server-side tool-call telemetry: every `tools/call` invocation the server has recorded into the config DB's `tool_calls` table - cross-client (corpus runs, foreign adopters), not this client's transcript.

When to use. To see how often each tool is actually used, how expensive and reliable it is, which clients and protocol eras are in play, and how much of the active tool pool the recorded traffic covers.

Parameters

Parameter	Type	Required	Default	Meaning
`topTools`	`int`	no	`30`	Cap on the per-tool breakdown, ranked by call count; clamped to 1-200.
`sinceDays`	`int`	no	`null`	Only count calls from the last N days (omit for the whole log); clamped to 1-3650. Narrows every figure (counts, percentiles, error classes, facets). The applied cutoff comes back as `windowSinceIso`, so an empty window is never mistaken for an empty table.

Read the scope first. The telemetry filter sits on the `tools/call` pipeline only. A sub-query inside `batch` or `measure` records the parent call and no child row of its own - except that a recent-enough parent row carries a per-tool tally of what it dispatched, reported under `subQueries` (see below). A one-shot `aicb call` writes the same row under the client name `aicb-call` when it is given `--db-path`, and nothing without one. Every count here is therefore a **lower bound**, and a tool at 0 was not called *through this door* rather than not called. The response repeats this scope in `recordedVia`.

Response. JSON:

Field	Content
`available` / `note`	`false` plus an explanatory note when the server is DB-free or the telemetry table is not readable.
Totals	`totalCalls`, `errorCalls`, `errorRatePct`, `distinctTools`, `distinctSessions`, `clientSessions`, `firstTs`, `lastTs`, `clients`, `serverVersions`. `clientSessions` says how many MCP connections the recorded calls came from: `distinct`, `calls` (the calls that carry a connection id) and a `note` on what one id cannot separate - the subagents of one client session share their parent's connection, and `aicb call` and older rows carry no id. Omitted while no call in the window carries one.

Field	Content
`tools`	Per tool, ranked by call count: `tool`, `calls`, `errors`, `errorRatePct`, latency (`avgDurationMs`, `p50DurationMs`, `p90DurationMs`, `maxDurationMs`) and result size in characters (`avgResultChars`, `p50ResultChars`, `p90ResultChars`, `maxResultChars`).
`errorClasses`	Per tool with errors, a histogram of the exception *types* (e.g. `{"McpException": 3, "ArgumentException": 1}`). `McpException` is a guided failure the tool raised on purpose (expired session, unknown policy token); any other type is a defect suspicion worth chasing.
`argumentBindingErrors`	The subset of errors where the *caller* named an argument the tool does not have, so the call never reached the tool. Not derivable from `errorClasses` (the same `ArgumentException` is also what a guard inside a tool raises). Omitted when there were none.
`ignoredArgumentCalls`	Per tool: calls that answered although they sent an argument the tool does not have, so that argument had no effect - the success-side twin of `argumentBindingErrors`. Omitted when there were none.
`aliasApplied`	How often a tool was called with a parameter's accepted alias instead of its real name, i.e. how often callers were silently corrected. Omitted when that never happened.
`protocolVersions`	The share of calls per protocol revision.
`clientEras`	The (era × client) cross-tab from the client's own protocol-level identification, capped at 50 groups with `clientEraGroupsTotal` and `clientErasTruncated` disclosing the rest.
`preInstrumentationCalls`	Calls recorded before this instrumentation existed. A null protocol version in `protocolVersions` means exactly that - not an unknown client - and belongs in the denominator.
`facets`	How often each facet (the task axis of `prepare_task` / `pack_for_task` / `export_markdown`) was addressed. Omitted until a call has addressed one.
`poolCoverage`	How much of the *active* tool pool this door's traffic covers: `poolSize`, `poolToolsFired`, `poolToolsNeverCalled`, `neverCalledNote`, `outOfPoolCallsIncluded`, `outOfPoolTools`.
`subQueries`	The per-tool tally of sub-queries dispatched through `batch` (uncapped, unlike `tools`). A `measure` sub-query is left out (it runs the tool fully, but the caller consumes the size of the answer, not the answer), and a sub-query naming a tool the dispatch registry does not know is counted under the single literal `(unknown)`. Omitted while nothing has been dispatched.
`windowSinceIso`	The applied `sinceDays` cutoff.

Notes and limits

- Read coverage from `poolCoverage`, not from `distinctTools`: that figure counts every tool called, including ones outside the pool, so holding it against the pool size overstates coverage. A name in `poolToolsNeverCalled` was never called *top-level*, which is not the same as never called - hence `neverCalledNote`. The note also names the span the list was measured over - the window cut, the oldest recorded call in it and how long ago that was - because a zero over one day and a zero over months are different statements. An empty window says the span is unknown: the list is unmeasured, not unused.
- The percentiles are the honest read for the right-skewed latency distribution: one warm-up call drags the average.
- This report is **not sufficient on its own for a prune decision**: a tool dispatchable through `batch` / `measure`, or reached by `aicb call` without `--db-path`, can be in real use and still read 0 here.
- Shapes only: no code content, no argument values, no exception *messages* (the type name only), and the session reference is hashed.
- The per-turn client-side view (tool sequences, how often a symbol question went to text search instead) lives in the agent harness's own transcripts, which this server never sees; the two windows differ.
- Not dispatchable through `batch` / `measure`.

`architecture_overview`

Purpose. Orient on a whole solution *without* the firewall of full source. Renders a structural-only AI-Builder-Markdown overview: the macro/architecture graphs (layer map, service and class dependency graphs, role graph, entry points and entry-point flow, architecture flow, interface relations), a domain summary and a quality-hotspots section (the most complex methods and the most-coupled types) - but no per-type or per-file source-code blocks, so it stays bounded where `export_markdown` renders the entire solution.

When to use. First contact with an unfamiliar codebase; before a structural change; whenever you need the shape of the solution rather than one symbol. Afterwards, drill in with `get_context` or `explain_symbol`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`lean`	`bool`	no	`true`	Drop the `<AI_CONTEXT_SPEC>` / `<AI_CONTEXT_META>` format-rules preamble and any empty graph sections. `false` renders the full AI-Builder-Markdown document contract.
`includeTests`	`bool`	no	`false`	Include test projects.
`summaryOnly`	`bool`	no	`false`	Collapse ordinary macro sections to a count plus top-N entries instead of up to 80. The layer map becomes an aggregate with the total and one count per layer.

Response. Markdown. Macro sections are capped at 80 entries with a `+N more` note. The layer map has its own bound: it lists every type up to 2000 types; above that it shows the total and one count per layer - never a sample - and its header names the bound and points to `export_markdown`, which keeps the full map. This keeps even a 200+-project monolith bounded. Method-level call graphs are deliberately omitted (use `call_graph` or `explain_symbol` for a specific symbol).

The `<TRUNCATION>` block. When anything was cut, the document leads with a `<TRUNCATION>` block: how many sections were capped, how many entries are shown of how many in total, and one `SECTION: shown of total` row per capped section - a layer map reduced to counts appears as `LAYER_MAP: 0 of N`. Read it before the sections: each `+N more` line states one section's loss, never the whole. No block means nothing was cut.

Notes and limits

- **Production-focused.** Test projects are excluded by default, because an architecture overview is otherwise flooded by every test method listed as an entry point plus its flow trace, which is not part of the production architecture. Set `includeTests: true` to include `*.Tests` / `*.Spec` projects in the graphs.
- Use `summaryOnly: true` on extreme solutions (200+ projects, heavy generics) where even the bounded overview would exceed the token cap and would otherwise return nothing.
- Multi-TFM solutions are deduplicated to one logical project per name, keeping the newest target framework's instance - otherwise every hotspot and edge would appear once per target framework.
- Recall-safe: works on live *and* recalled sessions. On a recalled session the quality-hotspot line counts are unavailable; complexity is still shown.
- Note: if every project in the solution classifies as a test project and `includeTests` is `false`, the production-focused view has no projects left, so every section derived from types (the macro graphs, entry points, quality hotspots, domain components) is empty by construction - not because the codebase has no architecture. A leading note says so and points to `includeTests: true`. If that classification is wrong, `solution_config_status` reports the test axis, and on a live session `init_solution_config` and `apply_solution_config` set it.
- Dispatchable through `batch` / `measure`.

`describe_api_surface`

Purpose. List the public API surface of a solution or namespace - every externally visible type with its externally visible members (constructors, properties, fields, events, methods, user-defined operators), each as a compact declaration signature. The "header"/contract view: what a consumer of this assembly can actually reference, without the source bodies.

When to use. To review a public contract before changing it, to see what an assembly really exposes, or as the baseline for a contract comparison (save a snapshot with `save_session` and compare later with `compare_with_previous`; the dedicated contract-diff tools are outside the default pool).

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`scope`	`string`	no	`"solution"`	`"solution"` covers everything; a namespace prefix (e.g. `MyApp.Contracts`) narrows the surface.
`includeInternal`	`bool`	no	`false`	Additionally surface the internal family (`internal` / `private protected`) types and members.
`includeTests`	`bool`	no	`false`	Include test projects.

Response. JSON: `scope`, `includeInternal`, an optional `note`, `typesScanned` and `types` (a capped envelope; the true total travels in the envelope). Each type carries `name`, `fullName`, `kind`, `accessibility`, `namespace`, `baseType`, `interfaces` and `members`; each member carries `kind` and `signature`. The type list is capped at 200; members per type are complete.

Notes and limits

- A type is included when its *own* declared accessibility is externally visible - `public` or the protected family (reachable by a subclass in another assembly); its members likewise.
- A property renders only the accessors that are themselves visible (`{ get; private set; }` shows as `{ get; }`); a `const` carries its value; an enum is listed as a type (its values are not in the model).
- Events follow the visibility rule of properties and render as their declaration, for example `public event EventHandler? Changed`: an interface event is listed, an explicit interface implementation is not.
- A `scope` prefix that matches no namespace gets a `note` ahead of `typesScanned: 0`, so an empty surface is not read as "nothing is public". The note says where to look instead: the namespace of a type the prefix names, the namespaces that contain the prefix as whole segments, or the closest namespace.
- A `partial` type is folded into one entry: members and interfaces of its declaration fragments are unioned, and a partial method's defining and implementing declarations count once. A multi-targeted type is listed once.
- Two documented simplifications: accessibility is the type's own declared modifier - the effective accessibility through the nesting chain is not computed, so a public member of a type nested in an internal type is still listed; and a type declared with no access modifier is stored as private, so a modifier-omitted (compiler-internal) top-level type is not surfaced even with `includeInternal: true`.
- Production-focused: test projects are excluded by default, because a shipped API surface is production code.
- Recall-safe: reads persisted accessibility and signature facts, so it works on live and recalled sessions.
- Dispatchable through `batch` / `measure`.

6.3 Sessions and analysis

`analyze_solution`

Purpose. Analyze a C#/.NET solution with Roslyn and cache it for the session. Returns the `sessionId` used by the other tools.

When to use. Call it explicitly when you want the analysis up front, want to control which configuration applies (layer profile, config DB), or want the returned session metadata. It is optional: every session-taking tool accepts the absolute solution path directly and analyzes the solution on first use.

Parameters

Parameter	Type	Required	Default	Meaning
`solutionPath`	`string`	yes	-	Absolute path to the solution file to analyze (e.g. `C:\src\MyApp\MyApp.sln`).
`layerProfile`	`string`	no	`null`	Absolute path to a layer-mapping profile JSON.
`dbPath`	`string`	no	`null`	Absolute path to a config/master DB.

`layerProfile`**.** Omit it to use the config DB's active layer profile (per-solution over app-global) when a config DB is resolved, else the `.aicb.json` sidecar's profile, else role-based heuristics. If the file you pass parses but contains no layer-mapping profile at all (no rules, no id, no name - the fingerprint of a schema-foreign file such as the `.aicb.json` sidecar passed by mistake), the call fails with an actionable error instead of silently degrading to heuristics. If you meant the sidecar, omit `layerProfile` - its layer profile is resolved automatically. Precedence: explicit profile > DB > sidecar > heuristics.

`dbPath`**.** When given:

- the DB's active namespace-exclusion list is applied during analysis (the same as the GUI and CLI);
- the DB's active test-detection profile (per-solution > global > default) is resolved for the session, so the test-aware tools honor it;
- and - unless an explicit `layerProfile` is given - the DB's active layer-mapping profile drives the analysis (dependency-graph layers and the layer map), matching the insights/metrics/CLI-gate path.

The layer and test profiles travel with the session, so downstream tools do not need `dbPath` again. Namespace exclusions travel only when they came from the DB-free sidecar path; exclusions supplied by a database affect the initial analysis but are not retained for a later full refresh. Omit `dbPath` to use the server's default config DB (the same one the GUI uses), if the server resolved one. On a DB-free server the defaults apply: no exclusions, the default test heuristic, role-based layers.

Response. JSON with these fields:

Field	Content
`sessionId`	The id to pass to the other tools.
`solutionPath`, `solutionName`	The analyzed solution.
`projectCount`	Number of projects.
`fileSetHash`	The file-set hash used by `refresh_session` to detect changes.
`layerProfile`	The layer profile that applied.
`analyzePreferredTfmOnly`	Present only when `true`: the `.aicb.json` asked for the preferred target framework only, so `projectCount` counts one instance per logical project instead of one per target framework.
`lastAccessUtc`	Last access timestamp (ISO 8601).
`origin`	`Live` (fresh Roslyn run, line numbers intact) or `Recalled` (rehydrated from a stored snapshot, no live workspace, no line numbers).
`lineNumbersAvailable`	Whether line numbers are available for this session.
`configInit`	Present when a configuration axis (layer / exclusions / test) has not yet been initialized through the guided setup.
`agentWiring`	Present only when there is something to say about the calling agent's symbol-guard installation.

`configInit` carries `layerProfileNeedsInit`, `exclusionListNeedsInit`, `testProfileNeedsInit`, `anyUninitialized`, `hint`, the per-axis `autoInitLayer` / `autoInitExclusions` / `autoInitTest` flags (omitted while `false`), and a `directive` when at least one axis is both opted into auto-initialization and still uninitialized. Note the polarity: these flags mean *needs init*, while `solution_config_status` reports the same three axes as *initialized* - the opposite sense. The hint points to `solution_config_status`, then `init_solution_config` and `apply_solution_config`; the result is persisted to the DB and to the `.aicb.json` sidecar next to the `.sln`.

Notes and limits

- **Progress.** A client that sends a progress token receives progress notifications during a long analysis. The progress channel is bound by the protocol, not a parameter you pass.
- A load failure is translated into an actionable message rather than a generic invocation error. The most common cause is a project that targets a newer SDK/TFM or a platform SDK (Windows SDK, Windows App SDK, Aspire) that the analyzer's build host cannot load.
- **What is read.** Every C# source file of the loaded projects, committed `*.Designer.cs` and `*.g.cs` files included; files in build output (a project's `obj/` and `bin/` folders and its intermediate and output directories) are not. A file named `AssemblyInfo.cs` is not read as source; the assembly-level attributes it applies still count as type references. A folder named `bin`, `obj` or `.vs` that *contains* the project - a checkout under `~/bin/repo`, say - is the checkout's location, not build output, so such a solution is analysed normally on every operating system.
- **Platform.** The projects are loaded at the platform the sidecar key `designTimeBuildPlatform` names, if the solution file names it; without it, a solution that defines no `Any CPU` platform is loaded at the platform its default configuration builds its C# projects at, when they agree on one. `get_diagnostics` reports which (see there).
- Writes session state only - no files, no database. Not dispatchable through `batch` / `measure`.

`refresh_session`

Purpose. Re-analyze the session's solution after *your* code edits. Without it, tools such as `find_usages`, `impact_of_change`, `list_insights` or `get_context` may keep answering from the stale pre-edit graph - a silent source of wrong results.

When to use.

1. You changed `.cs` files.
2. Call `refresh_session`.
3. Continue with the fan-in, quality or context tools.

Also call it with `force: true` after a restore or build that was meant to repair unresolved references.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`force`	`bool`	no	`false`	Re-analyze even when no source file has changed.

Response. JSON: `changed`, `reason`, `session` (the same session metadata block `analyze_solution` returns) and `mode`. `mode` names the path the re-analysis took: `"incremental"` (document texts replayed - and added or removed C# files applied - into the warm snapshot, no MSBuild reload) or `"full-reload"`. It is absent when nothing was re-analyzed.

Notes and limits

- Cheap to call speculatively: a file-set hash check skips the expensive Roslyn re-run when nothing actually changed, and a C# file whose timestamp moved while its text stayed the snapshot's (a save without an edit) does not count as a change either.
- An edit of a `.cshtml` or `.razor` file counts as a change too: the call reports `changed: true` and re-analyzes with a full reload.

- An added, renamed or removed `.cs` file is applied to the live solution without an MSBuild reload when the files around it show which project compiles it; `mode` is then `incremental` and `reason` says `applied to the live solution, no MSBuild reload`. Where that is not clear, or when a `.xaml`, `.razor`, `.cshtml` or project file was added or removed, the full reload runs and the reason says so. An ordinary full reload (not `force`, not a restore) also reuses what it can, and `reason` says how many documents were reused from the previous run.
- That check reads *source* files. A restore that rewrote a project's `project.assets.json` after the session was loaded is seen as well: the call then reloads through MSBuild and its `reason` names the project. A build alone is not seen. After one, an analysis that reported unresolved references retries once by itself; `force: true` is the deterministic way to demand it. It costs a full MSBuild reload, so leave it `false` for the normal after-an-edit refresh.
- `mode` is the only indicator when the incremental path stops engaging: answers stay correct, calls just get slow again.
- Note: when nothing was re-analyzed but the session's last analysis was incomplete, the answer carries a note explaining that a restore/build repairs that without touching a source file, and telling you to call again with `force: true`.
- Reuses the warm workspace. A recalled session has no live workspace and is rejected with a message pointing to `refresh_remembered` or `analyze_solution`.
- Writes session state only - no files, no database. Not dispatchable through `batch` / `measure`.

`get_diagnostics`

Purpose. A fast pre-build gate: the compiler errors and warnings of the session's solution in seconds instead of a full `dotnet build`.

When to use. After editing `.cs` files, as a quick correctness check before you build or run anything. The recommended order is `refresh_session` first, then `get_diagnostics`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`severityFloor`	`string`	no	`"auto"`	Minimum severity to include: `"auto"`, `"error"`, `"warning"`, `"info"` or `"hidden"`. `"auto"` counts warnings as `"warning"` does, but when the compile has errors it lists only the errors, and `warningsNotListed` says how many warnings it left out of the list; `"warning"` lists both.
`scope`	`string`	no	`"solution"`	`"solution"` or a file-path substring (e.g. `MyApp.Core` or `Services/`) that filters the result.
`runSuppressors`	`bool`	no	`false`	Run the DiagnosticSuppressors that the projects' analyzer references declare, as a real build does, so a diagnostic they suppress is not counted. Off by default, because a suppressor is code of the analyzed solution and runs inside the server; the answer's `suppressors` member says where it would matter.

Order matters. The diagnostics are compiled from the *session*'s document snapshot, not from the files on disk. Under auto-refresh mode `off` they describe the pre-edit source - a freshly written error can answer `errorCount: 0`. Under the shipped `reactive` mode this tool is refreshed for you before it answers, so the explicit `refresh_session` call is redundant rather than wrong; calling it is the spelling that is correct in *every* mode. An answer computed from a graph known to be behind the disk leads with `verdict: "stale"` and `verdictReason` before any count, so reading the pre-edit state deliberately stays possible - it just cannot be mistaken for a clean bill.

Response. JSON with these fields:

Field	Content
`verdict` / `verdictReason`	Lead before any count when the answer is not a clean measurement. `"inconclusive"` when a strict majority of scanned projects are incomplete; `"stale"` when the session's graph is behind the disk - edited source files, or a project whose restore rewrote its `project.assets.json` after the session was loaded (the reason names the project; call `refresh_session`, then ask again). If both hold, the wider negation (`inconclusive`) is the single-token verdict and the reason names both.
`scope`, `severityFloor`	The echoed request; under the default `auto` the echoed floor reads `"warning"`, the floor the counts are taken at.
`projectsScanned`	How many projects were compiled.
`errorCount`, `warningCount`, `infoCount`	Counts over the full, uncapped set.
`suppressedDiagnosticsTotal`	How many diagnostics the incomplete projects took with them. Present whenever at least one project is incomplete.
`warningsNotListed`	Under the default `auto` only, when the compile has errors and warnings: `count` (the warnings counted in `warningCount` but not listed) and a `note` naming `severityFloor: "warning"` to list them; the disclosures, `generatedCodeGaps` among them, still count those warnings. Placed right before `diagnostics`. Omitted whenever the list is the full one.
`diagnostics`	The diagnostics, capped at 200 - under the default `auto`, only the errors when there are any; the counts above reflect the full set. Each carries `id` (e.g. `CS0219`), `severity`, `message`, `location` (file:line), `category` and `project`.
`incompleteProjects`	Projects whose compilation could not resolve its core references: project name, suppressed-diagnostic count, and `affectedDependents` (the projects that transitively reference it and inherit the broken chain).
`incompleteRatio`	The share of incomplete projects among the scanned ones.
`cascadeClassifiedCount`	How many of the listed diagnostics are cascade-classified fallout of an incomplete dependency (each such item carries `cascadeFromIncomplete: true`).
`referenceBindingAdvisoriesSuppressed`	How many assembly-reference version-float advisories (`CS1701` / `CS1702`) were filtered out.
`reliable`	`false` when a strict majority of scanned projects are incomplete.
`generatedCodeGaps`	Code that a source generator or the WPF markup compiler writes is missing from a project. A generator reference that points at the output of a solution project built at another configuration or platform than this load builds it at is named as that - kind `not-built`, or a hedged `not-found` where it cannot be told whether the project holds a generator - not as an unrestored package; where the producing project declares no source generator, the reference is not listed at all.
`designTimeBuildFailures`	MSBuild reported a failure loading a project (a warning reads alike); its compiler options may be incomplete. A Visual Studio Web Site entry in a `.sln` (a URL or folder path with no project file) is in most cases not counted as a failure (such a solution is then cached, so its next open is fast); it still is when a loaded project lost a project reference; a missing `.csproj` stays a failure, and Web Site entries in a `.slnx` are not recognized.
`unrestoredProjects`	Projects that resolve their core references but have unresolved-name errors (`CS0246` / `CS0234`) and no `project.assets.json` where their restore writes it: `unresolvedNameErrors`, a `note`, and `projects`, each with `project`, `searchedFrom` and `unresolvedNameErrors`. The note names the repair - `dotnet restore`, and on `NETSDK1147` `dotnet workload restore`. Nothing is removed from the counts. Omitted when no such project exists.

Field	Content
`uninitializedSubmodules`	Git submodules that were never initialized (their folder is empty) while a project with `CS0246` / `CS0234` errors names a path inside one in its build files (the project file, an import, the nearest `Directory.Build.props` / `.targets`): `unresolvedNameErrors`, a `note`, and `submodules`, each with `path`, `projects` and `unresolvedNameErrors`. Neither a restore nor a build brings those sources; the note names the repair, `git submodule update --init <paths>` in the repository root, then `refresh_session(force: true)`. The notes of `unrestoredProjects` and `generatedCodeGaps` add a sentence when their projects are among these. Omitted when no such submodule exists.
`missingCompileItems`	Compile items that a project with counted errors lists while no file exists at that path, so the compilation read each as an empty file - typically a file a pre-build step or a code generator writes that did not run: `errors`, a `note`, and `projects`, one entry per project file (every target framework of a multi-targeted project in one), each with `projectFile`, `projects` (the instance names), `files` (solution-relative, in full outside the solution folder) and `errors`. The note says which errors are likely this and the repair: run the build step, then `refresh_session(force: true)`. Omitted when no such item exists.
`designTimeBuildPlatform`	The platform decision of the load: `platform`, `solutionConfiguration`, `solutionPlatforms`, `projectsAtOtherPlatform`, `blockers` and a `note`; `source: "sidecar"` when the sidecar decided, and `refusedSidecarPlatform` when the sidecar value was refused. When the sidecar names a solution platform that maps the projects to another platform name (for example `Win32` built at `x86`), the `note` says so. Present when the load chose a platform, could not choose one, or refused the sidecar's value; `platform` is absent when no single platform could be chosen. Omitted when the solution defines `Any CPU`, builds every C# project at AnyCPU, or could not be read, and the sidecar names no platform. See the note below.
`suppressors`	Present when a DiagnosticSuppressor removed a diagnostic, or when an analyzer reference that declares one (NUnit.Analyzers, EF Core and the Razor compiler are common ones) did not run its suppressors in a project the answer counts a diagnostic in; omitted otherwise. `ran` says whether `runSuppressors` was set; `diagnosticsSuppressed` counts what the suppressors removed (per target framework); `notRun` names each reference whose suppressors did not run, with a `reason` and its `projects`; `diagnosticsInAffectedProjects` counts the diagnostics in those projects, where a real build may report fewer - a magnitude, not a verdict; `note` says what each half means. Without `runSuppressors`, the references are found by reading their metadata only, which runs none of their code.

Notes and limits

- `#pragma`**-suppressed diagnostics are excluded**; `[SuppressMessage]` on a compiler warning changes nothing, as in a real build. Diagnostics that a DiagnosticSuppressor removes in a real build are counted unless you pass `runSuppressors: true` (see `suppressors`). A multi-targeted project that compiles the same source under several target frameworks reports each diagnostic once (deduplicated by id + location + message), not once per framework.
- **Incomplete projects are disclosed, not listed.** A project whose compilation cannot resolve its core references (`System.Object` missing - a targeting pack not installed for that TFM, or an unrestored project) would otherwise flood with thousands of bogus cascade errors. It is not listed as findings but appears under `incompleteProjects`. A freshly created, never-built worktree reports *every* project there until it is built once - that is the unrestored state, not a defect, and one `dotnet restore` / `dotnet build` makes the fast gate work for the rest of the session.
- **Design-time build platform.** A solution that defines no `Any CPU` platform is loaded at the platform its default configuration builds its C# projects at, when they agree on one; where they do not, each project is loaded at its own default platform (AnyCPU for an SDK-style project), the `note` says why, and `blockers` lists the projects that prevented a choice where there are such projects. To set the platform yourself, put `"designTimeBuildPlatform": "x64"` into the solution's `.aicb.json` sidecar - a platform as the solution's configuration manager names it, such as `x64`, `ARM64` or `Any`

`CPU`. A value the solution file does not name (as a solution platform, or as a platform it maps a C# project to), or one under which the solution maps its projects to several platforms, is refused and disclosed, and the automatic choice applies. The value is read on every full reload, so after editing it call `refresh_session` with `force: true`; an unforced refresh after an edit of the sidecar alone keeps the platform the session was loaded at. Where failures or missing generated code may come from the platform, the notes say so and name a build of the affected projects with `-p:Platform=<platform>`.

- The `inconclusive` verdict is deliberately rare: it fires only when a strict majority of projects are incomplete, because a judgment that triggers on every partly-restored solution is one you learn to ignore. The *disclosure* is not rationed the same way: whenever even one project is incomplete, the answer carries `incompleteRatio` and `suppressedDiagnosticsTotal` next to the counts.
- **Read `suppressedDiagnosticsTotal` as a magnitude, not as one half of a ratio with `errorCount`.** The two are counted under different rules: the total spans every severity from the floor up and is *not* deduplicated (a multi-targeted incomplete project contributes once per target framework), while `errorCount` counts errors only and *is* deduplicated across target frameworks. A total that dwarfs `errorCount + warningCount + infoCount` means this measured a fraction of the solution, even when no verdict is present.
- A core-resolved project whose dependency chain includes an incomplete project inherits the broken references and can list cascade ids although it compiles fine on a real build. Those diagnostics are never hidden: they stay listed, marked `cascadeFromIncomplete: true`, and counted in `cascadeClassifiedCount`, so `errorCount` is not read as "N real errors".
- Assembly-reference version-float advisories (`CS1701` / `CS1702`, which carry no source location) are binding-redirect noise from the analysis host's reference resolution and are never actionable from source; on core-resolved projects they are filtered out and their count disclosed.
- `scope` is a **post-filter, not a narrowing**: every project is still compiled and `projectsScanned` reports all of them, so latency is the same as a solution-wide call. Use `scope` to save response tokens, never to save time - for one quick check after an edit, one solution-wide call is as cheap as a scoped one.
- An unknown `severityFloor` is rejected with the list of valid values, `auto` included.
- The default `auto` keeps the answer of a large failing edit small: the list holds the errors, not the warnings. The cap still counts items, not characters, so 200 very long errors can still make a large answer.
- **Live-only:** diagnostics are not persisted, so this tool needs a live `analyze_solution` session; a recalled session is rejected (use `refresh_remembered` for a live one).
- Compiler diagnostics only - third-party Roslyn *analyzer* diagnostics are out of scope. The one opt-in exception is `runSuppressors: true`, which runs the projects' DiagnosticSuppressors (they remove compiler diagnostics, they add none). Loading an analyzer assembly for that also runs the constructors of the analyzers it declares, inside the server.
- Dispatchable through `batch` / `measure`.

`inspect_session`

Purpose. Metadata about *one* cached session.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.

Response. The same session metadata block `analyze_solution` returns: session id, solution path and name, project count, file-set hash, layer profile, last access, origin (`Live` / `Recalled`) and line-number availability.

Notes and limits

- Not exposed by default. It belongs to the session plumbing group that no facet menu and no bundle names, so it is absent from `tools/list` and a direct call is refused unless the server is started with the `AICB_MCP_TOOLS` override naming `SessionTools`, for example:

`text AICB_MCP_TOOLS=SessionTools`

- Not dispatchable through `batch` / `measure` either.

`list_sessions`

Purpose. List all currently cached analysis sessions (id, solution path, project count, last access).

Parameters. None.

Response. One entry per cached session, in the same shape as `inspect_session`.

Notes and limits

- Not exposed by default, for the same reason as `inspect_session`: only the `AICB_MCP_TOOLS` override reaches it. Not dispatchable through `batch` / `measure`.

6.4 Finding symbols, signatures and single-symbol context

`find_symbol`

Purpose. Find types, methods, properties, fields, events, enum members and operators whose name contains the query (case-insensitive).

When to use. Locate a symbol before calling `find_usages`, `call_graph` or `get_context`, or to disambiguate a name for `explain_symbol`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`query`	`string`	yes	-	A substring of the symbol name to search for (type, method, property, field, event, enum member or operator).

Aliases. The tool schema advertises `query`, but `symbol` and `name` are accepted as stand-ins and are rewritten to `query` (a recorded caller confusion, kept as a tolerance). An explicit `query` always wins over an alias if you pass both.

Response. JSON: `items`, `count`, `totalFound`, `truncated`, on a zero-hit result `nearest`, and, where it applies, `generatedSourcesNote` - on a zero hit, and on a populated answer when the name is also a Razor component written only in markup (see below). Capped at 100. Each item carries `kind` (`type`, `method`, `property`, `field`, `event`, `enum_member` or `operator`), `name`, `declaringType`, `namespace` and `signature`, plus `project` where two hits would otherwise read alike - for example the same type declared under the same full name in two projects, which are kept as two hits.

Result order. Results are ranked by match quality first - exact name, then case-insensitive exact, then prefix, then substring - and only within one rank by kind order. An exact match is therefore never hidden by the cap behind weaker substring hits. If the result is truncated, the exact and prefix matches are the ones you got; narrow the query (a longer substring) to see the weaker rest.

Kind order. Types are listed first, then methods, then properties (handwritten and source-generated; a generated property carries a `// source-generated` note in its signature), then fields, events and enum members, then user-defined operators and conversions. An enum member's signature is its qualified `Enum.Member` form - the string the fan-in tools take.

Notes and limits

- A user-defined operator is found by its *metadata* name: query `op_Equality`, `op_Addition` or `op_Implicit`, and `op_` lists them all. Every kind listed here is one that `find_usages` and `impact_of_change` resolve; an operator resolves by the same metadata name, with its uses - `a == b`, a cast, an implicit conversion - as its fan-in.
- Not indexed, so an empty answer is expected for them: local variables, parameters, labels and namespaces; C# generated into `obj/` that carries real declarations (protobuf/Grpc.Tools, T4, Refit); and a Blazor component written only in a `.razor` file. A zero-hit answer says which of these may apply in `generatedSourcesNote` - the `.razor` case names the

component's file - and, unless the query names a `.razor`-only component, also names a source generator that did not run in the analysis, whose output could hold the declaration.
- A bare name that is both a C# member and a Razor component written only in markup (a constant `Snippets.AlertSimpleExample` beside a component `AlertSimpleExample.razor`) finds the member; when no C# type of that name is in the answer, `generatedSourcesNote` adds that the name is ALSO such a component, whose class has no symbol and no fan-in here, and names its file.
- When the query matches nothing, `nearest` names the closest declared symbol - a likely typo or case mismatch. A matched symbol never carries it.
- Dispatchable through `batch` / `measure`.

`symbol_signature`

Purpose. Return a symbol's signature(s) *without* the body - the type declaration, the method or operator signature, or a member's declaration (property, field, event, enum member, the last as its qualified `Enum.Member` form) - plus its XML `<summary>` documentation and its declaring file and start line.

When to use. Understand an API without reading the whole file or body, and navigate straight to the declaration.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`symbol`	`string`	yes	-	The exact (case-sensitive) type, method, property, field, event, enum-member or operator name. A member also takes `Type.Member` (that type's member only), a type a qualified name (`Ns.Type`, `Outer.Inner`). A user-defined operator matches by its metadata name (e.g. `op_Equality`).

Response. A capped envelope (`items`, `count`, `totalFound`, `truncated`), capped at 50, plus `nearest` when nothing matched and, where it applies, `generatedSourcesNote`. Each item carries `kind`, `name`, `declaringType`, `namespace`, `signature`, `summary`, `line` and `file`. One declaration compiled in two projects is two results, each with its `project`.

Notes and limits

- `file` is the fragment's *own* file: for a partial type, a method points at the file the method lives in, not at the first type fragment. It is omitted only when the snapshot carries no path.
- `line` is the line the declaration starts on, a leading attribute included - for types, methods, operators, properties, fields and events alike. It is `null` for an enum member and for a source-generated property, for which no line fact is modeled.
- Matching is exact, so a typo or case mismatch returns an empty list rather than an error; the `nearest` suggestion distinguishes "you spelled it differently" from "this symbol genuinely has no signature". Where the run skipped generated C# under `obj/`, a component lives only in a `.razor` file, or a source generator did not run, `generatedSourcesNote` names that third possibility: the declaration exists in code this index never read. Multiple results are returned for overloads or name collisions.
- A member that shares its bare name with a Razor component written only in markup is returned as usual, and `generatedSourcesNote` names the component as a different symbol, as in `find_symbol`.
- A qualified member query respects every segment: `Outer.Inner.Member` or `Ns.Type.Member` answers for the member of the type whose full name ends in that chain, also through a generic outer type (`Outer<T>.Inner.Save`), not for every type that carries the last type segment's name.
- Leaner than `get_context`, which returns the full source.
- Dispatchable through `batch` / `measure`.

`get_context`

Purpose. Return a dense, token-budgeted AI-Builder-Markdown slice of the code around a symbol: the symbol's source plus its expanded neighborhood (direct dependencies and callees, depth 1). The fastest single-symbol retrieval.

When to use. Whenever you need one symbol's context. Prefer it over `export_markdown` (which renders the *whole* solution). If you need a chosen environment of one symbol (callers, tests, implementations, quality), use `explain_symbol`; to bundle a whole task from a natural-language goal, use `pack_for_task` or `prepare_task`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`symbol`	`string`	yes	-	The type or method simple name to center the slice on. Use `find_symbol` first to disambiguate.
`budget`	`int`	no	`null`	Token budget, floored at 8000.
`includeQualityMetrics`	`bool`	no	`false`	Also emit the `<QUALITY_HOTSPOTS>` section plus `complexity="..."` / `ce="..."` tag attributes.
`lean`	`bool`	no	`true`	Drop the `<AI_CONTEXT_SPEC>` format-rules preamble, the `<AI_CONTEXT_META>` block and empty graph sections.

`budget`**.** Caps the slice in two ways: it compacts or drops method bodies, and - when the expanded neighborhood still overflows - it drops the least-relevant non-seed types entirely, ranked by closeness to the seed and in-slice fan-in. The seed symbol itself is never dropped: it renders with its full source when that fits the budget, otherwise as *structure* (identity, members, every method signature) with a leading note naming the measured size of the full-code render and the budget that would hold it. A type reduced to structure lists the signatures of every method it declares that does not already render as a method block, up to 80 lines and then a `(+N more)` line, under a comment such as `<!-- N declared methods, signatures only: the budget took this type's source out (see the budget note) -->`; the list counts against the budget. Without a value the slice still renders against a ceiling - the active MCP profile's token budget, else a default of about 10,000 tokens - that bounds the neighborhood. Pass a value for a tighter slice.

`includeQualityMetrics`**.** Set it when reviewing code quality or picking refactor targets: the slice additionally carries cyclomatic complexity and efferent coupling, as a `<QUALITY_HOTSPOTS>` section and as tag attributes on the sliced symbols. The afferent coupling (`ca=""` on a type tag, and the hotspot coupling line) counts the referencers in the whole solution, not only those the slice kept, so it is the number a full export prints.

`lean`**.** On a focused single-symbol slice the format-rules boilerplate is more than four times the actual signal, which is why it is dropped by default. The `PATH_LEGEND` and `COMPRESSION_LEGEND` (the decoding keys) and all non-empty sections are kept. Set `lean: false` for the full AI-Builder-Markdown document with the format contract.

Response. Markdown. A leading comment discloses how many types were dropped.

Notes and limits

- Multi-TFM solutions are deduplicated to one logical project per name, keeping the newest target framework's instance - the same view `find_symbol` lists. `export_markdown` deliberately keeps the unfiltered per-target-framework render.
- When several types share the name you pass (in different namespaces), every one of them renders, not only one.
- A slice centered on an interface pulls its implementing types - those that implement it directly, through a base class or through another interface, for a generic interface whatever type arguments they close it with. Under a budget the leading comment names how many types were dropped (`dropped N least-relevant types ...`).
- A partial type renders as one block per project it is compiled into: within each project its declaration fragments are folded, and a leading comment says which files. Where the type is compiled into more than one project (a linked file, a shared project), that comment names the project it was folded in (`<Type> in project <Name>`), also when the other project holds only a single fragment of it. The type name is namespace-qualified when another entry of the same comment has the same simple name - for example when two projects fold it.
- Dispatchable through `batch` / `measure`.

`explain_symbol`

Purpose. Explain a symbol: its source *plus* a chosen environment, as one dense AI-Builder-Markdown slice - the cold read in a single call instead of four to six (a context slice plus the fan-in, implementation and test queries).

When to use. When you need to understand one symbol and its neighborhood at once: who calls it, what it calls, who implements it, which tests cover it.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	`string`	yes	-	Session id or absolute solution path.
`symbol`	`string`	yes	-	The type or method simple name, or the qualified `Type.Member` form.
`include`	`string[]`	no	`null`	The environment axes to include; any combination.
`budget`	`int`	no	`null`	Token budget, floored at 8000; behaves as in `get_context`.
`lean`	`bool`	no	`true`	As in `get_context`.

`symbol`**.** A bare name that matches several declarations seeds them *all*, and every type that shares the name renders; the manifest says so and names the qualified form to use instead. The qualified `Type.Member` form focuses one declaration of a shared method name - the same string `find_usages`, `find_tests_for` and `impact_of_change` resolve.

`include` **axes** (case-insensitive):

Axis	What it adds
`callers`	The members that reference the symbol, with their bodies, and their owner types as structure - not the calling types whole. A caller that references the symbol only from its structure (a field or property type, a base list) renders as that caller's own declaration.
`callees`	What the symbol calls or depends on, depth 1. This is the widest slice this server produces.
`implementations`	The types implementing it, if it is an interface - also those that get a generic interface through a base class or another interface, as `find_implementations` lists them. Every same-named type the axis lists renders.
`tests`	Its covering test cases. For a method, the method's own tests, bare or qualified - not the tests that read a same-named property. Strongest evidence first, at most 40; the manifest discloses a cap as `prepare_task` does (`(+N more, capped)`, with the share of contract rows when there are any).
`siblings`	Naming and file-convention kin. Every same-named type the axis lists renders.
`quality`	Complexity and efferent-coupling hotspot metrics.

Response. Markdown with a leading *manifest* comment that discloses what each requested merge axis actually found - so `(none)` is an honest negative (no implementation, no caller, no test), not "not asked". A partial type renders as one block per project it is compiled into (its declaration fragments are folded; a leading comment says which files, and names the project where the type is compiled into more than one) instead of one identical block per file.

Other kinds of the same name. The callers axis answers for the center - the type or the method the symbol resolved to. When a bare name also names a property, field, event, enum member or operator elsewhere (for a type center, also a method), those members' users are not counted as the center's callers: a center nobody uses reads `callers: (none)`, and without the callers axis the manifest no longer states their fan-in as the center's. The manifest then carries a `namesakes:` segment, for example:

```
namesakes: 'Clock' also names property TimeService.Clock (3 direct user(s)) - their users are not counted as
the type's callers; find_usages takes the qualified Type.Member form to measure one
```

It names up to three forms per kind and then `and N more`, with each kind's direct user count, appears with or without a callers axis and also when the center has users of its own, and is absent for a qualified query and when no other kind of the name has users.

Notes and limits

- An empty `include` returns just the symbol's source; the manifest then names the axes you could have asked for, so that answer is a stated choice rather than a silent default. An `include` token that names no axis is reported as dropped, whether or not a sibling token was valid.
- `budget` behaves as in `get_context`: the center symbol is always kept - with its full source when that fits, otherwise as structure plus a leading note naming the measured size of the full-code render and the budget that would hold it. The bundled environment (callers, tests, siblings, implementations) is budget-bound and dropped to fit under a tight budget; a leading comment names the drops so you can ask for one by name. Without a value, the slice renders against the same ceiling as `get_context`.
- Recall-safe: reads persisted facts, so it works on recalled sessions. Prefer it over `export_markdown` for one symbol's world.
- Multi-TFM solutions are deduplicated to one logical project per name, keeping the newest target framework's instance (matching `find_symbol`'s view).
- Dispatchable through `batch` / `measure`.

7 Tool reference: impact, hierarchy, tests and DI

The tools in this chapter answer four groups of questions:

- **Impact and fan-in** - who uses a symbol, how far a change propagates, which call or dependency path connects two symbols, and who constructs a type.
- **Hierarchy and lifecycle** - who implements an interface, who overrides a method, how a type's inheritance tree looks, and how a type is created, injected, returned, referenced and persisted across the solution.
- **Tests and coverage** - which tests cover a symbol, which methods no test calls directly, which complex methods are untested, and whether a negative claim about a symbol holds.
- **Dependency injection** - which concrete type is registered for an interface (by Microsoft DI or as a MEF export), with which lifetime, and where.

7.1 Conventions used in this chapter

- **Session.** Every tool takes a `sessionId`. Most tools also accept the absolute path to a `.sln` file directly and analyze it on first use. A tool answer is computed from the analyzed session, so after you edit source files call `refresh_session` before asking again - otherwise the symbol graph keeps answering from the pre-edit state. See "Sessions and staleness".
- **Read-only.** All tools in this chapter only read the analyzed model. None of them changes source files, the session, or any database.
- **Recalled sessions.** All tools in this chapter work on a session restored from a saved snapshot, except `resolve_injection`, which needs a live analyzed session.
- `note`**.** Where a response carries a `note`, read it before the numbers. It leads the answer when the answer needs a qualification - an empty result that is not proof of absence, a run whose project references did not all resolve, or a zero that is not a measurement. Notes are omitted when there is nothing to say.
- **Caps.** Capped lists report the true total in `totalFound` and set `truncated` to `true`. Narrow the query or the `scope` to see the rest.
- **Availability.** Each tool below carries an availability line. "Core" tools are part of every tool set. "Default" tools are part of the default tool set. A tool marked "not in the default tool set" is still shipped, but you reach it only when the server runs with an MCP profile whose facet menu includes it (for example the Exploration, Debugging or Architecture facet), or when you add its name through the `AICB_MCP_TOOLS` environment variable.
- **Parameter aliases.** Some tools accept a familiar alternative spelling for their main argument, listed in the parameter tables as an alias. The canonical name is always advertised; the alias is only tolerated so a call that used the other name still binds.

7.2 Fan-in, impact and paths

`find_usages`

Purpose. List who uses a symbol - the fan-in. This is the central "who calls X?" tool and the first call for almost every impact question.

When to use it. Before renaming, moving or deleting a symbol; when you need the concrete list of callers, readers, writers or subscribers; when you want to know whether a type is referenced at all.

Availability. Core tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id returned by `analyze_solution`, or an absolute `.sln` path for self-init.

Parameter	Type	Required	Default	Meaning
`symbol`	string	yes	-	Exact (case-sensitive) name of a type, method, property, field, event, enum member or user-defined operator (by its metadata name, such as `op_Equality`) - or the qualified `Type.Member` form to disambiguate a member. `Type.Type` asks for a type's constructor.
`includeTests`	bool	no	`true`	Include users declared in test projects. `false` lists production users only, before the 100-item cap, and `testUsersFiltered` says how many were left out. A user no fact places in a project stays listed. `viaContract` keeps counting every caller.

What a name resolves to

Query	`resolvedKind`	Answer
Type name	`type`	The types that reference it - a type of the same simple name that derives from it, or holds or creates it under another arity or a namespace-qualified name, included (`Identity<T1, T2> : Identity`, `Ef.User : Core.User`), while a type that names only itself is not its own user - plus markup views, `.razor` files, `.cshtml` views and files of top-level statements. A doc `cref` counts (unmarked), and so does an attribute application at any position; an assembly- or module-level attribute is credited to the file that applies it, as `<file> (assembly)` or `<file> (module)`. A generic type parameter that shares the type's name (a `T` beside a solution type `T`, a Razor `@typeparam`) is not a use of it.
Method name	`method`	The methods that call it (the union across overloads).
Property or field name	`property` / `field`	The methods that read or write it. A CommunityToolkit `[ObservableProperty]` backing field additionally lists the consumers of its generated property - the two are one symbol pair to a caller.
Event name	`event`	The methods that raise, subscribe to or unsubscribe from it. The subscriber types are also available through `find_by_event_subscription`.
Enum member (bare `Partial` or qualified `BackupStatus.Partial`)	`enum_member`	The methods that reference it - a switch label, a comparison, an assignment.
User-defined operator or conversion by its metadata name (bare `op_Equality` or qualified `Money.op_Implicit`)	`operator`	The code that uses it - `a == b`, `a + b`, `++`, `+=`, a cast and an implicit conversion of a value - in method, accessor, constructor, initializer and operator bodies. A use through a generic constraint lands on the interface's static abstract operator, not on the implementing type's. A record's compiler-synthesized `==` has no declaration and answers `not_found`.
`Type.Type`	`constructor`	The type's construction and injection sites, a `new` in top-level statements included (named by its file label). This analyzer records no per-constructor fan-in, so overloads are unioned and a listed site may have called a different overload than you meant; `instantiation_sites` reports the same facts with the two edge kinds kept apart. `impact_of_change` does not resolve this form.

Qualified members. A qualified member form respects every segment: `Outer.Inner.Member` or `Ns.Type.Member` answers for the member of the type whose full name ends in that chain - also through a generic outer type (`Outer<T>.Inner.Save`) - not for every type that carries the last type segment's name. The simple form `Type.Member` still answers for every type of that name, and says so in `namesakes` when there are several.

Where the users come from. Besides C# code, the fan-in includes:

- `.cshtml` **views** (MVC, Razor Pages) that the Razor source generator compiled in the analysis: what a view calls, reads or names is credited to the view file, labelled `<path>.cshtml (razor)`. A type injected in a `_ViewImports.cshtml` is credited to that file once, not to every view below it, and an `@attribute [X]` in a view is a use of the attribute type. Views the

generator did not compile are named on a zero answer (see "Empty answers").
- **Top-level statements.** Every type a `Program.cs` without a class names counts as used by that file, labelled `<path>/Program.cs (top-level)`: `typeof`, type arguments (`AddTransient<I, Impl>()`), casts, `is` / `as` and patterns, `default(T)`, `nameof(T)`, declared locals, `out`, `foreach` and `catch` types, tuple elements, attributes on local functions, a record `with`, collection expressions, a creation, and the types in a called method's signature.
- **Markup** in the forms listed under "Empty answers" below.

Example

```
{ "sessionId": "<session>", "symbol": "OrderService.Cancel" }
```

Response. `symbol`, `resolvedKind`, an optional `note`, and `usedBy` - a capped envelope (`items`, `count`, `totalFound`, `truncated`, maximum 100) - and, on a production-only answer (`includeTests: false`), `testUsersFiltered`. Each item is an object `{name, file, line}`, one per user, so `totalFound` counts users:

- `name` - the user: a calling member under its full key with parameter types, a referencing type, or a markup view.
- `file` - the file the user is declared in, relative to the solution and with forward slashes.
- `line` - the line the user's declaration starts on in that file (its first attribute line when attributes precede it), not the line of the reference. A field, property or event whose initializer uses the symbol is placed by that member's declaration.

A partial type declared in several files lists them in `files` as `[{file, line}]` objects instead of `file` and `line`. Where one name stands for several users - two types of one simple name in different namespaces, the same type or caller compiled in two projects - each declaration is its own item under the shared `name`, and the items add `fullName` (for a type), plus `project` where the full name or the caller is shared too. A lone item carries neither - except a user whose name is the queried name itself (a generic `Foo<T>` listed as a user of `Foo`), which names its `fullName`. Markup and top-level entries carry a `file` but no `line`, and an item that no recorded fact places carries neither: absent means unknown, never a guess. A script written for earlier releases, whose items were bare strings, reads `name`.

```
{
  "symbol": "OrderService.Cancel",
  "resolvedKind": "method",
  "usedBy": {
    "items": [
      {
        "name": "MyApp.Web.OrdersController.Cancel(int)",
        "file": "src/MyApp.Web/OrdersController.cs",
        "line": 41
      }
    ],
    "count": 1,
    "totalFound": 1,
    "truncated": false
  }
}
```

Depending on the answer, additional steering fields appear:

- `nearest` - the closest declared name, set when the query matched nothing (`resolvedKind: "not_found"`). A likely typo or case mismatch; it can also be a declared name the query only begins, or - for a member asked on a type that does not declare it - the one type that does.
- `collision` - set when a bare name also names a symbol that is not counted: for a member answer a same-named type, or - where no C# type has the name - a Razor component written only in markup (that component has no symbol here and its users are not counted); for a member or type answer, another member kind of the name that has users (named

with its qualified forms and direct user count).
- `namesakes` - set when the name resolved to a type that the solution declares several times (a generic and a non-generic type of one name, such as `Foo` and `Foo<T>`, count as two and are named in their declaration form), or a qualified member whose type part still names several types (`'<query>' is declared on N DIFFERENT types named '<Type>' ...`, with a further-qualified example). `usedBy` is then the union over all of them, and no entry says which one it uses. A bare member name keeps its union without this note.
- `viaContract` and `viaContractNote` - set when a qualified method was credited the callers of the interface or abstract members it implements.
- `selfReferences` - how many of the listed users of a member sit on its own type. Counted only when the matched members sit on one type by full name; two same-named types are not taken for one owner.
- `textuallyInvisibleUsers`, `textualBindingKinds`, `textualBindingNote` - see below.
- `testUsersFiltered` - on `includeTests: false` only: how many users the filter left out because they are declared in test projects. For the constructor form (`Type.Type`) test sites are dropped too, and the note says how many construction edges sat in tests. `selfReferences` and the textual-binding fields are counted over the users that remain.

`not_found` **is not "undeclared".** A delegate type and a qualified or nested type name (`Outer.Inner`) answer `not_found` too. Where the declarations can explain the miss, a `note` says why: a member asked on a type that does not declare it (the note names the types that do), a qualified member whose qualifier names none of the same-named types that declare it (the note names their full names and suggests a tail of one of them, or `Type.member` for all of them), a declared type named by a qualified form, namespace-qualified or nested (the note names the simple name to ask for), or a Blazor component written only in a `.razor` file (the note names its file; it does not list the component's users).

Empty answers. A real kind with an empty `usedBy` means the symbol exists but no intra-solution reference was recorded. That is not proof of dead code, and the `note` that leads the answer names the ways a real use can escape the index and a cross-check you can run. On `includeTests: false`, an answer whose users were all in test projects says so instead (`No PRODUCTION user was recorded: all N recorded users are declared in test projects ...`) and keeps the same list of escapes:

- **Markup, beyond the recorded forms.** Recorded in XAML are `{Binding}` root paths in both markup-extension and element syntax (low-confidence, tagged `(xaml)`, attributed to the resolved view-model); event wires in `.xaml` and Avalonia `.axaml` alike - an attribute that names an event the element's type or one of its bases declares, or an attached `Owner.Event`, credited to its handler, a `void` method with two parameters (where the element's type cannot be resolved, only `Click` counts; an `.axaml` wire is labelled `<file>.axaml (axaml)`); property sites in `.xaml` and `.axaml` - an attribute or property element that sets a property, an attached `Owner.Member` (also under a URI-mapped prefix), and the `Property` of a `Setter`, `Trigger` or `Condition`, credited to the property, or to `SetX` where an attached property is set and `GetX` where it is read. A bare `Setter` name is read against the enclosing `Style` or `ControlTemplate` `TargetType`, or against the type the last segment of an Avalonia `Style Selector` names (`Button` in `Button.accent` or `Button:pointerover`, a `ns|Type` through its prefix), a nested `^` style inheriting its parent's; a selector naming several types or none (a comma, only a class, a name or `:is()`), the `[Property=]` conditions inside a selector, and a type the run could not resolve record nothing, and an Avalonia `Classes.accent="True"` toggle is not a use of a `Classes` member. Also recorded: `{x:Static}` members, attached-property read accessors, `{TemplateBinding}` members and markup type references, a nested type written `prefix:Outer+Inner` included (in `{x:Type}`, `x:Static`, `DataType`, `TargetType`, `d:DesignInstance`, attached-property owners and markup extensions). Recorded in `.razor` files are members named in `@code` blocks, in code-block statements and in component-parameter values (a value without `@` only where the parameter's type is known and is not `string`), and a call through an injected member (`@inject`, `[Inject]`) or through the component's own field or property, credited to the member on that member's type. A WPF view with neither a design hint nor a matching name takes the one view-model type that production code assigns to its `DataContext` (for example `new MainWindow { DataContext = shell }`); assignments in test projects do not count, and a view assigned two different types gets none. A view-model reached only through a `d:DesignInstance` design hint or the `FooView` -> `FooViewModel` naming convention records its member edges but not its type edge, so a view-model rename can still read as low-risk.
- **C#** `nameof` of a method outside a method's own attributes (a `nameof` in a method's attribute, such as `[RelayCommand(CanExecute = nameof(CanSave))]`, counts as a call; a `nameof` of a type is recorded).
- **Activation by name at runtime:** reflection, a container resolving by convention.

- **Source-generated code:** generated documents are not walked, so a generator's output records no edge on what it calls - except the code the Razor generator compiled from a `.cshtml` view, which is read and credited to the view's file. Committed `*.Designer.cs` and `*.g.cs` files outside build output are not affected: they are read like any other source file.
- `.cshtml` **views the Razor generator did not compile** - a runtime-compiled view in a plain `Microsoft.NET.Sdk` project, a generator that did not run, or views the design-time build did not hand to it. A zero answer names them: `VIEWS NOT READ: <n> .cshtml view(s) in <projects> were not compiled by the Razor generator in this run (<m> other view(s) were read) - ...`, with the advice to search the name in those projects' `.cshtml` files before reading the zero as unused. A non-zero answer does not carry it.
- **Enum members only:** values that arrive as strings or numbers (`Enum.Parse` / `TryParse`, JSON or database deserialization, a XAML attribute literal, a command-line argument) and attribute arguments on a type, property or field.
- **Events only:** a subscription made through an interface lands on the interface's event, not the class's (the bare name unions both); a handler added by name (`WeakEventManager.AddHandler(src, nameof(E), ...)`, reflection `AddEventHandler`) or wired in markup leaves no edge.
- **Operators only:** a `dynamic` operand, bound at run time; a call by name through reflection; the operator `true` or `false` behind a user-defined `&&` or `||` ; the element operators of a tuple comparison `(a, b) == (c, d)` ; and a conversion that a `foreach` applies to its elements, a compound assignment to its operand or result, or `??` to its right side.

The cross-checks the note suggests: for a member, run the fan-in query on its declaring type (a type's fan-in is recorded where its member's is not); for a type, ask `instantiation_sites`, which reads different facts (construction and injection edges); for either, search the name in `.xaml`, `.csproj` or `.json`, or read the source.

Textually invisible users. When the answer has entries that bind to the symbol without naming it anywhere in their own source - a value consumed from a member that returns the type, a call that omits an optional parameter of the type, a call to a member inherited from a generic base class closed over the type, or a call to or an override of a member inherited from the type through another base class - the response reports the count in `textuallyInvisibleUsers` and splits it in `textualBindingKinds` :

Kind	Meaning	Breaks when
Via return type	Consumes a value of the type from a member that returns it.	The type's contract changes (a member is renamed or removed); survives a type rename.
Via optional parameter	Calls a method that takes the type as an optional parameter and omits the argument.	Only when that method's signature changes; survives a rename and a contract change.
Via closed generic base	Calls a member inherited from a generic base class that its own type closes over this type (`OrderRepository : RepositoryBase<Order>`), and holds no value of it.	Only when that base list changes; survives a rename and a contract change of the type.
Via inherited member	Derives from the type through another base class and calls a member the type declares (bare, `this.` or `base.`) or overrides one.	When those members change; survives a rename of the type.
Unclassified	Binds by a mechanism this check does not recognize.	Inspect by hand.

The JSON names of the kinds are `viaReturnType`, `viaOptionalParameter`, `viaClosedGenericBase`, `viaInheritedMember` and `unclassified`. Every kind is present in `textualBindingKinds`, at 0 where it does not apply, and the kinds add up to `textuallyInvisibleUsers`. A call to an inherited member on another object (`other.Save()`) is left unclassified: the check cannot tell it from a namesake. `textualBindingNote` states the same in one sentence. Only a count is reported, not a per-entry flag.

Limits. Matching is case-sensitive. The list is capped at 100; on a heavily tested symbol test users can take most of those slots, so ask with `includeTests: false` for the production fan-in. A member name resolves by simple name across all types unless you use the qualified `Type.Member` form.

Run-wide qualification. If the analysis run could not resolve every project's references, a `note` leads every answer of this family - including a populated one - because such a run is short by ordinary references, not only exotic ones. Rebuild and call `refresh_session`, then repeat the query.

A generator that did not run. When the symbol is declared in a project where a source generator or the WPF markup compiler did not run, or in a project that depends on one, the answer carries `generatedCodeNote` - on a populated answer too, because the list is then short by ordinary references. See "Recurring conventions across the catalog".

`impact_of_change`

Purpose. Estimate the blast radius of changing a symbol: the direct and the transitive fan-in (who depends on it, directly or through chains) plus a risk level. This is the call to make before you refactor or rename shared code.

When to use it. Before an edit that changes a symbol other code names - a rename included. The trigger is the symbol's reach, not the kind of edit.

Availability. Core tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`symbol`	string	yes	-	The same strings `find_usages` accepts - a user-defined operator by its metadata name included - except the constructor form `Type.Type`, which is not resolved here (use `find_usages`). A type takes its simple name only.

Response. `symbol`, `resolvedKind`, an optional `note`, `directCount`, `transitiveCount`, `risk`, `productionImpactCount`, `directImpact` (capped envelope, maximum 20 entries), the textual-binding fields described under `find_usages`, and `nearest` on an unresolved name. Where they apply: `saturation`, `viaContract` and `viaContractNote`, `implementers`, `riskBasis`, `collision` and `generatedCodeNote` (ahead of `risk`; see "Recurring conventions across the catalog").

Field	Meaning
`directCount`	Number of direct users, counted one per declaration exactly as `find_usages` lists them (its `totalFound`). Includes test-project callers.
`transitiveCount`	Number of dependents in the full transitive closure, one per declaration: where several types or callers share a name, each declaration is its own node, and a dependent is followed to the declaration it refers to where that can be told. Includes test-project callers.
`productionImpactCount`	The non-test part of the transitive closure.
`risk`	`none`, `low`, `medium` or `high`.
`directImpact`	The direct users, as items `{name, file, line}` exactly as in `find_usages` (a type declared in several files lists `files` as `[{file, line}]`).
`saturation`	Set where one dependency cycle dominates a high closure: it leads the counts, names the cycle, and risk is then graded by the direct production fan-in.
`viaContract`, `viaContractNote`	A qualified method also counts the callers of the interface or abstract members it implements, as in `find_usages`.
`implementers`	On an interface or abstract method or type: one entry per contract with `contract`, `count`, `testCount`, `typeCount`, `testTypeCount` and `members` - what a signature change breaks. See below.
`riskBasis`	Set together with `implementers`: `closure`, `implementers`, `total` and an optional `note` - the number the risk was graded on.

Field	Meaning
`collision`	A same-named symbol of another kind, not counted.

```
{
  "symbol": "OrderService",
  "resolvedKind": "type",
  "directCount": 1,
  "transitiveCount": 2,
  "risk": "low",
  "productionImpactCount": 2,
  "directImpact": {
    "items": [
      {
        "name": "CompositionRoot",
        "file": "src/MyApp.Console/CompositionRoot.cs",
        "line": 9
      }
    ],
    "count": 1,
    "totalFound": 1,
    "truncated": false
  }
}
```

A script written for earlier releases, whose `directImpact` items were bare strings, reads `name`.

What is counted, per kind. A type: the closure of the types that reference it. A method or an operator: its caller closure, following interface and abstract dispatch at every hop. A creation counts as a call to the constructor it binds, so for a method called from a constructor the closure reaches the code that builds the object, and its risk can rise. A property, field, event or enum member: the code that touches it - there is no transitive member chain, so `transitiveCount` equals `directCount`. A field or event initializer in a test class counts as test code, not as a production consumer.

How the risk level is calibrated. The thresholds are `none` for 0, `low` from 1 to 4, `medium` from 5 to 20, and `high` above 20. They read `productionImpactCount`: test-project callers are excluded, because exercising a symbol from tests is its designed fan-in and updating those tests after the change is mechanical, not risk. Two cases read a different number:

- With `saturation`, the bands read the direct production fan-in the block names, because a count that every symbol reaching the cycle shares measures the cycle, not the symbol.
- With `implementers`, the bands read `riskBasis.total` = `riskBasis.closure` (the caller count above) + `riskBasis.implementers` (the production implementing types), because a signature change breaks both. On a method every production implementing type counts. On an interface or abstract class only the types that name it in their own base list count, and only those the closure does not already hold.

The risk counts dependents, not what the change means.

Implementers. An `implementers` entry for an interface or abstract **method** lists the implementing and overriding members; `typeCount` counts the types they sit on, one per declaration (two same-named types in different projects count twice), which is what a change to one overload breaks. When the member name is declared by two same-named interfaces or abstract classes in different namespaces, the entry keeps their combined count and `namesakes` names each contract with its own implementing types. An entry for an interface or abstract **type** has one entry per declaration (namesakes in different namespaces and generic interfaces of different arity stay apart) and lists the implementing or deriving types, including those that implement the interface through a base class or another interface - for a generic interface whatever type arguments they close it with, at any depth - as `find_implementations` lists them; the ones that reach it only through a base class, and the interfaces that extend it below the

first level (their own base list names a derived interface, not this one), are listed again under `indirect` and do not raise the risk - for a generic interface as for a non-generic one. A class that gets the interface through a derived interface is not indirect: it implements the members, so a signature change breaks it. A type in the global namespace and a namespaced type of the same simple name are not taken for each other in these counts. When no production code calls the interface or abstract member - or, for a type, no production type references it - `riskBasis.note` says that the risk comes only from its implementing types; the risk level itself is unchanged by the note. Where `saturation` grades the closure by its direct users, its note names the implementing types added to that count.

Notes.

- `risk: "none"` on a solution that has production projects does not by itself carry a note for a symbol used only by tests: `directCount` is then greater than 0 while the risk is `none`. That class is reported by `find_production_dead`, which is outside the default tool set.
- On an unresolved name the answer leads with a note stating that the numbers measure nothing and that `risk: "none"` is the absence of a measurement, not a safe verdict. A mistyped name and a genuinely dependency-free symbol produce the identical answer. Where the declarations explain the miss, the note says why, as in `find_usages` - including a Blazor component written only in a `.razor` file.
- A resolved symbol with `directCount` 0 carries the escapes note described under `find_usages`, including the `VIEWS NOT READ` sentence when `.cshtml` views were not compiled. The users `find_usages` lists from `.cshtml` views and top-level statements count here as well, and a qualified member resolves as described there.
- If every project in the solution classifies as a test project **and** `productionImpactCount` is 0, a note says so; read `directCount` / `transitiveCount` instead. The count is not structurally forced to zero: unresolved keys and markup fan-in labels can still be classified as production impact.

Example

```
{ "sessionId": "<session>", "symbol": "OrderService" }
```

`call_graph`

Purpose. Build a depth-limited call graph rooted at a method name.

When to use it. To see the shape of a call neighborhood - what a method calls, or who calls it - without reading bodies. For a single path between two methods use `trace_flow`.

Availability. Core tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`method`	string	yes	-	The root method's name. Alias: `symbol`.
`depth`	int	no	`2`	Maximum traversal depth. Values are clamped to 1-10.
`direction`	string	no	`"callees"`	`"callees"` follows project-internal calls outward; `"callers"` follows the inverse (who calls this).

Response. `root`, `direction`, `maxDepth`, `edges` (pairs of caller and callee keys), `truncated`, and `rootedAt`. The edge list is cycle-safe and capped at 500 edges; `truncated` says whether the cap was hit.

Notes.

- The graph is rooted at the name, not at one symbol: every same-named declaration in the solution is a root and the graph is their union. A common name such as `ExecuteAsync` can pull in unrelated members. Check `rootedAt` - the declarations actually walked - when the result looks too large, and re-query a rarer name.
- An empty `rootedAt` means the name resolved to nothing; an empty edge list alone does not tell you that.
- Note: `direction` is not validated. Any value other than `callees` selects the `callers` direction.

Example

```
{ "sessionId": "<session>", "method": "RunAsync", "depth": 3, "direction": "callers" }
```

`instantiation_sites`

Purpose. Find who instantiates a type - its construction and dependency-injection sites. This is the narrow "who creates or obtains an instance of X?" answer that `find_usages` (every reference: field, parameter, generic argument and so on) cannot give.

When to use it. When a type's constructor changes and you need the call sites that must be updated; when you want to know whether a type is obtained through the container rather than constructed; when a fan-in query on the concrete type returned a surprisingly small list.

Availability. Default tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`typeName`	string	yes	-	The type's simple name. Namespace and generic arguments are normalized away. Alias: `symbol`.
`scope`	string	no	`"solution"`	A namespace prefix that narrows which sites are scanned.

Response. `typeName`, an optional `note`, `createdByCount`, `createdByTests`, `injectedIntoCount`, `injectedIntoTests`, `sites` (a capped envelope, maximum 200 entries; created sites first, then injected), and `nearest` when the name matched no declaration. Each site carries `name`, `declaringType`, `namespace`, `kind` and `isTestProject`.

Edge kind	Meaning
`created`	A method, constructor, accessor or top-level code that runs `new X(...)`. This includes target-typed `new`, `record with`, and collection or `stackalloc` allocation. A field or property initializer `= new X()` is attributed to the constructor the compiler runs it in - the implicit one when none is declared - so such a site is named `.ctor`; with several non-chaining constructors, one initializer line yields one site per constructor. A creation in top-level statements is named by its file label (`<path>/Program.cs (top-level)`) with no declaring type and no namespace, so a namespace `scope` leaves it out; a multi-targeted project's file is one site.
`injected`	A type that receives X as a constructor-injected dependency; the container constructs X and hands it over. For these sites `declaringType` is null, because the consuming type is the site.

Notes.

- Test-project sites are included, because a test that constructs X breaks when X's constructor changes.
- A concrete type registered in a dependency injection container reports `injectedIntoCount: 0` correctly: its consumers take the interface, so the injection edges sit on the interface's name. When the injected count is zero and the type implements interfaces, the note names them and points you to the query that carries the answer (`resolve_injection` confirms the registration).

- When both counts are zero, a note names the construction paths this fact cannot see: a container registration whose consumers take an interface, a generic service-locator call such as `GetRequiredService<T>()` or `GetService<T>()` (the type appears as a type argument, which is neither edge kind; a fan-in query on the type names the calling type, and `resolve_injection` counts such calls per service under `otherConsumers`), and `Activator.CreateInstance` or other reflection.
- On a session recalled from a snapshot stored before top-level creations were recorded, the answer carries `TOP-LEVEL CREATIONS NOT KNOWN` (see "Recall safety" under "Recurring conventions across the catalog").
- The tool matches the name against facts without resolving it, so a misspelled type or one outside a narrowed scope yields the same 0/0. When `typeName` matches no declaration, the construction-boundary note remains and a `nearest` suggestion appears alongside it.

Example

```
{ "sessionId": "<session>", "typeName": "OrderRepository", "scope": "MyApp.Infrastructure" }
```

`trace_flow`

Purpose. Trace the shortest static call path between two methods.

When to use it. To understand how a flow connects two methods, or which chain of callers triggers a method, before editing.

Availability. Not in the default tool set; reachable through a profile whose facet menu includes it (for example Exploration, Debugging, Documentation, Architecture or Performance), or via `AICB_MCP_TOOLS`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`from`	string	yes	-	The anchor method's name (the start of the trace).
`to`	string	yes	-	The target method's name to reach.
`maxDepth`	int	no	`12`	Maximum traversal depth. Values are clamped to 1-50.
`direction`	string	no	`"callees"`	`"callees"`: how does `from` end up calling `to`? `"callers"`: through which caller chain does `to` reach `from`? Only the exact value `callers` (case-insensitive) selects the reverse direction.

Response. `from`, `to`, `direction`, `found`, `length`, `path` (an ordered `Type.Method` list from `from` to `to`), and a `note`. When no path exists within `maxDepth`, `found` is `false` and the note says so. In `callers` mode each step is called by the next. `length` counts the nodes on the path, not the hops, so a direct call reports `length: 2`.

Notes. Resolution is name-based. The tool follows only project-internal static call edges - not interface dispatch, events, dependency injection or reflection. It reads persisted call facts and works on recalled sessions.

Example

```
{ "sessionId": "<session>", "from": "Main", "to": "SaveOrder", "maxDepth": 20 }
```

`type_dependency_path`

Purpose. Trace the shortest type-dependency path between two types - the type-level counterpart to `trace_flow`. It answers the *why* of a transitive dependency, where `impact_of_change` gives only the set of impacted types.

When to use it. To find the chain that connects two types, for example why a domain type transitively reaches an infrastructure type - a layering smell.

Availability. Not in the default tool set; reachable through a profile whose facet menu includes it (for example Exploration, Debugging, Documentation, Architecture or Performance), or via `AICB_MCP_TOOLS`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`from`	string	yes	-	The anchor type's simple name (the start of the trace).
`to`	string	yes	-	The target type's simple name to reach.
`maxDepth`	int	no	`12`	Maximum traversal depth. Values are clamped to 1-50.
`direction`	string	no	`"dependencies"`	`"dependencies"`: how does `from` end up depending on `to`? `"dependents"`: through which dependent chain does `to` reach `from`? Only the exact value `dependents` (case-insensitive) selects the reverse direction.

Response. `from`, `to`, `direction`, `found`, `length`, `path` (an ordered type-name list from `from` to `to`) and a `note`. When no path exists within `maxDepth`, `found` is `false` and the note says so. In `dependents` mode each step is depended on by the next. `length` counts the nodes on the path, so a direct dependency reports `length: 2`.

Notes.

- A type A depends on B when A references B: as a base type or interface, field, property, parameter, return type, generic argument, or a type used in its body. These are the same forward type-dependency edges that `find_usages` inverts and `impact_of_change` closes over, so a `dependencies` path from A to B exists exactly when A appears in `impact_of_change(B)`.
- Resolution is name-based on simple type names; cross-namespace name collisions collapse. Only intra-solution types can be endpoints - external and framework types are not.

Example

```
{ "sessionId": "<session>", "from": "OrderService", "to": "SqliteOrderRepository" }
```

7.3 Hierarchy, implementations and lifecycle

`find_implementations`

Purpose. List the types that implement an interface, whether the interface is declared in the solution or external (`IDisposable`, `IEquatable`).

When to use it. Before changing an interface, to see every implementer that must be updated; when you want to know whether an interface has one production implementation or many.

Availability. Core tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.

Parameter	Type	Required	Default	Meaning
`interfaceName`	string	yes	-	The interface's simple name. An arity suffix in metadata or declaration form - `IConsumer` 1 `or` IConsumer `, or` IConsumer `0` for the non-generic namesake - filters to that arity; a bare name keeps the merged view over all arities. A concrete type argument (`IConsumer<OrderPlaced>`) keeps only the types that close the interface with that argument. Case-sensitive; a namespace-qualified name matches nothing, and two interfaces sharing a simple name answer as one list. Aliases: `symbol` , `interface` .
`includeTests`	bool	no	`false`	Include test-project implementers.
`typeArgumentBase`	string	no	`null`	Keep only the types that close the generic interface with a type argument that is this type, or derives from or implements it - a simple name, case-sensitive, with an arity suffix for a generic (`DeletedNotification<T>` , `ICancelableNotification`). Only solution types form the family; when none matches, `typeArgumentBaseFilter` carries a note.

Response. An optional leading `note` (set when the analysis run could not resolve all project references), the item list in the shared capped envelope (`items` , `count` , `totalFound` , `truncated` , maximum 100, sorted by name), an optional `hint` , an optional `generatedCodeNote` , `testImplementationsFiltered` , `typeArgumentFilter` on a closed query, and `typeArgumentBaseFilter` when `typeArgumentBase` is set.

Each item carries `name` , `isAbstract` , `kind` (`class` , `struct` , `record` or `interface`), `isTestProject` and, where known, `namespace` , so an abstract base or a test fake is distinguishable from a concrete production implementer. A generic implementer additionally carries `declaration` , its arity-qualified form such as `PolicyWrap<TResult>` , so a generic and a non-generic pair sharing name and namespace stay distinguishable. `via` says how an item was found when it is not by an interface the type itself declares: `via: "back-edge"` marks an item known only from the interface's own implementer list, `via: "inherited"` a type that gets the interface through a base class or interface declared in the solution - including an interface that extends the queried one through another interface, at any depth.

On a generic interface each item also carries `closes` : the type arguments it closes the interface with, directly or through its bases and generic interfaces, one entry per distinct argument list (`ContentDeletedNotification` , or `Saved, IContent` for two arguments). An open generic implementer names its own type parameter; a position that cannot be read - an outer type parameter of a nested type - is written `*` . An item that does not name the interface in its own base list carries `through` : the base class or interface it gets it through (`IContentLastChanceFinder` , or `WebhookEventBase<TNotification>` for a generic solution type). Both are omitted when empty or when the chain could not be read.

Generic interfaces. A bare or arity query on a generic interface the solution declares (`IHandler` , `IHandler<T>`) lists every type that implements it, whatever type arguments it closes the interface with: those that declare it directly, and those that get it through a base class (generic or not) or another interface, at any depth below (`Update : HandlerBase<Saved, string>` , `Leaf : Update`), marked `via: "inherited"` . The same holds for a non-generic interface nested in a generic type (`Outer<T>.INested`). A type in the global namespace and a namespaced type of the same simple name are not taken for each other.

Closed queries. A query with a concrete type argument (`IHandler<Saved>`) lists the types that close the interface with those arguments, also through base classes and generic interfaces; such a walked item carries no `via` when the solution declares the interface. The closed answer is always a subset of the open one: the subclass of an unrelated same-named base class is not listed. A position written as a type parameter (`T` , or the name the interface itself declares there) is a placeholder, so `IHandler<T>` and the metadata form keep the arity answer. `typeArgumentFilter` reports `typeArguments` and counts the arity-matching types - test projects included - that were left out because they close the interface otherwise (`otherClosings`) or only over their own type parameters (`openClosings`), and how many stayed with an unreadable closing (`unresolved`). Such an item stays in the list, marked `closing: "unresolved"` .

Type-argument families. `typeArgumentBase` answers "which classes handle any notification of this family": it keeps the implementers that close the interface with an argument that is the named type or derives from or implements it, at any depth, including types that get the interface through a base class. Like the closed form it filters before the test filter. An implementer whose closing cannot be read stays in the list, marked `closing: "unresolved"` . `typeArgumentBaseFilter` reports `base` ,

`familyTypes` (how many solution types form the family), `leftOut` (the implementers no family argument closes, test projects included) and `unresolved`, and a `note` when nothing could match - the interface is not generic, or no solution type and no implementer names the base.

Notes.

- Test-project implementers are excluded by default. When that filter removed something, `testImplementationsFiltered` reports how many - a short list is never silently short. Set `includeTests: true` to see them.
- An empty result carries a `hint` in two cases: when the name is a type that is not an interface (the hint points to `get_type_hierarchy` and `find_overrides`), and when no type in the index carries the name at all (the hint says so and carries a did-you-mean when one is close enough; a delegate counts as undeclared). An empty list is a statement about the name, not about implementers.
- `generatedCodeNote` flags a source generator that did not run in the interface's project, as on the other declaration tools (see "Recurring conventions across the catalog"). On an empty result it also appears when a generator that did not run may have added the interface below it - in a project that uses the interface's project, or in any project for an interface the solution does not declare.

Example

```
{ "sessionId": "<session>", "interfaceName": "ICodeAnalyzer", "includeTests": false }
```

```
{ "sessionId": "<session>", "interfaceName": "INotificationHandler", "typeArgumentBase":
"DeletedNotification<T>" }
```

`find_overrides`

Purpose. List the override-to-base relationships for methods of a given name - the class-inheritance counterpart to `find_implementations`. It answers both "who overrides this virtual or abstract method?" (impact) and "what does this override?" (comprehension) in one call.

When to use it. Before changing a virtual or abstract method's signature, and when you want to see how a base behavior is specialized across the hierarchy.

Availability. Core tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`methodName`	string	yes	-	The method's exact (case-sensitive) simple name, for example `ToString` or `Dispose`. Alias: `symbol`.

Response. The shared capped envelope (`items`, `count`, `totalFound`, `truncated`, maximum 100). Each entry pairs an overriding method with the nearest ancestor that declares the virtual or abstract method it overrides, rendered as `Derived.M(params) overrides Base`. Container and base render in their declaration form, for example `Strategy<T>.M(...) overrides ResilienceStrategy<TResult>`. When the overridden base is a framework virtual outside the solution, the entry reads `... overrides (external base)`. An empty list means no overrides of that name were found.

Notes. Two same-named types that differ only by namespace, or that declare identical type-parameter names, render identically and collapse into a single row - the list carries no namespace. Treat a row as the relationship, not as one attributed type. Where a row stands for same-named types from different namespaces, `mergedNamesakes` follows the list and names each such row and its types. When the overriding type has the same simple name as its base, the row still names the base that declares the overridden member.

Example

```
{ "sessionId": "<session>", "methodName": "Dispose" }
```

`get_type_hierarchy`

Purpose. Walk a type's inheritance hierarchy in both directions: the base-class chain upward and the transitively derived types downward, plus the type's implemented interfaces.

When to use it. To see where a type sits in the hierarchy, which types derive from a base, or which interfaces a type implements. For "who implements this interface", use `find_implementations` instead - interfaces never appear as a base class, so an interface's `derivedTypes` is empty.

Availability. Core tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`typeName`	string	yes	-	The type's exact (case-sensitive) simple name. Add an arity suffix - `Repository<T>` or `Repository`1 `- to address a generic type unambiguously when a generic and a non-generic type share the name. Alias:` symbol`.

Response. `type`, `resolvedKind`, an optional `note` (the run-wide qualification), an optional `hint`, an optional `generatedCodeNote`, `mergedNamesakes`, `separatedNamesakes`, `mergedArities`, `baseChain`, `interfaces`, `derivedTypes` (a capped envelope, maximum 100) and, on `not_found`, a trailing `nearest`.

- `baseChain` lists the base-class chain upward, nearest base first. When the chain leaves the solution, the last entry names the external base, for example `ObservableObject (external)`. A base type outside the solution that only shares its simple name with a type inside it ends the chain as external.
- `interfaces` lists the interfaces the type itself declares - one inherited through a base class is not repeated; for an interface query these are its base interfaces.
- `derivedTypes` lists transitively derived types, one entry per type (partial fragments and the instances of a multi-targeted project stay one entry, and `totalFound` counts types); each entry reads `Derived : DirectBase`, so the subtree structure is preserved in the flat list. Test-project types are included and not marked.
- Every link is rendered arity-qualified, so a same-name generic and non-generic type are distinguishable. Types sharing a simple name are written by full name, for example `Ef.User : Core.User`, and where the full name is shared too - the same type in two projects - the project follows in brackets (`Extensions [MyApp.Web] : ExtensionsBase`). A link between two types of the same simple name is kept in both directions.
- `mergedArities` is set when a bare name matches a generic type and its non-generic namesake: the view merges them, and the field says whose base chain is shown.
- `resolvedKind: "not_found"` answers three empty lists: no type of that name is declared in the solution - also for an external base, a delegate type, a namespace-qualified name, a nested `Outer.Inner` form, or an arity suffix no declaration has. A trailing `nearest` then gives a close declared type name, for a qualified form the simple name. A Blazor component written only in a `.razor` file is named in the `not_found` note with its file, because its class has no symbol here.

Notes.

- The simple-name index merges namesakes - same simple name, different declared type, from a foreign namespace or a different declaring type. When the answered view merged such types, `mergedNamesakes` names them ahead of the lists they may affect. An arity suffix separates arities only; namesakes cannot be separated.
- `derivedTypes` lists a type only when its declared base really is the resolved type. A derivation from a same-named *other* type - typically a framework base shadowed by an in-solution namesake - is excluded and counted in `separatedNamesakes`, which names the resolved type and the foreign bases. Without it, an empty `derivedTypes` on a

name the solution plainly uses as a base would read as "nothing derives from this".
- An interface query carries a `hint` stating that `derivedTypes` covers class inheritance only and pointing to `find_implementations`.

Example

```
{ "sessionId": "<session>", "typeName": "GraphSectionRendererBase" }
```

`lifecycle_of`

Purpose. Map the full lifecycle of a type T in one edge-partitioned slice - who creates, injects, returns, references and persists it - instead of calling `find_usages` and `instantiation_sites` separately and partitioning by hand.

When to use it. When you need the whole picture of how a type is produced and consumed; and for the recurring "persisted but dropped" bug class: if the write leg shows up in `created` or `persisted` but nothing reads T back, that asymmetry is the smell.

Availability. Not in the default tool set; reachable through a profile whose facet menu includes it (for example Exploration, Debugging, Documentation, Architecture or Performance), or via `AICB_MCP_TOOLS`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`typeName`	string	yes	-	The type whose lifecycle to map. Simple name; namespace and generic arguments are normalized away. Alias: `symbol`.
`scope`	string	no	`"solution"`	A namespace prefix that narrows which sites are reported - not the target type.

Response. `typeName`, `scope`, an optional leading `note`, `targetIsDeclared`, `typesScanned`, `methodsScanned`, and the five buckets `created`, `injected`, `returned`, `referenced` and `persisted`. Each bucket is a capped envelope with its true total (maximum 100 entries), and each entry carries `name`, `declaringType` and `namespace`.

Bucket	Meaning
`created`	Methods, and files of top-level statements, that run `new T(...)`. A top-level file is named by its label (`<path>/Program.cs (top-level)`) with no declaring type and no namespace, so a namespace `scope` leaves it out.
`injected`	Types that receive T as a constructor-injected dependency.
`returned`	Methods whose return type is or wraps T - `Task<T>`, `IReadOnlyList<T>` and other factory shapes included.
`referenced`	The full type fan-in, equivalent to `find_usages`. This is the read/use umbrella and deliberately overlaps the specific buckets.
`persisted`	Best-effort: methods with a database or serialization side effect that reference T - the persist and round-trip edge.

Notes.

- The answer states what it looked at: `scope` echoes the applied filter, `typesScanned` is the population behind `injected` and `referenced` (a partial type counted once), and `methodsScanned` is the population behind `created`, `returned` and `persisted` - it also counts the top-level files whose creations were read (a file that creates nothing is not counted). Because `scope` narrows the sites of every bucket, it is a second route to five zeros that has nothing to do with the type.
- `targetIsDeclared` says whether a type of that simple name is declared in the solution at all. `false` means an external type or a misspelling, and any edges shown are then this solution's use of an outside type rather than that type's lifecycle.
- A `note` leads the buckets whenever a zero needs qualifying.

- Like `find_usages` and `instantiation_sites`, this is a discovery and impact tool, so test-project sites are included.
- It reads persisted facts and works on recalled sessions. A session recalled from a snapshot stored before top-level creations were recorded carries `TOP-LEVEL CREATIONS NOT KNOWN` on an unscoped query, also on a populated answer.

Example

```
{ "sessionId": "<session>", "typeName": "OrderDto", "scope": "MyApp.Application" }
```

7.4 Tests and coverage

`find_tests_for`

Purpose. Find the test cases that likely cover a symbol.

When to use it. Before changing a symbol, to see which tests pin it; when you want to know whether a method is exercised at all.

Availability. Core tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`symbol`	string	yes	-	A type name, or a member name either bare or in the qualified `Type.Member` form - the same strings `find_usages` and `impact_of_change` resolve, a user-defined operator by its metadata name (`op_Equality`) included - or `Type.Type` for the constructor.

Which methods count as tests. Only a method that itself carries a test attribute counts - an actual runnable test, not a fixture builder or helper that merely lives in a test project. The default attribute set is `Fact`, `Theory`, `Test`, `TestMethod` and `TestCase`, matched as a case-insensitive substring so that variants such as `WindowsFact` are recognized. The set is configurable through the solution's test profile.

Response. The shared capped envelope (`items`, `count`, `totalFound`, `truncated`, maximum 50), plus `invokesTotal`, `dispatchTotal` (omitted at 0) and an optional `note` after the list. Each item carries `testType`, `testMethod`, `project` and `matchReason`.

Match tiers. `matchReason` names one of nine tiers, in sort order:

Tier	Meaning
`invokes`	The test calls the target itself.
`invokes-via`	The test reaches the target through one method or constructor it calls - still strong, but second-hand. A test that builds an object reaches what that object's constructor calls.
`accesses`	The test reads or writes the target property or field; for a type, a static one.
`accesses-via`	The same, through one method or constructor the test calls.
`constructs`	The test builds the target type (type and constructor queries only).
`constructs-via`	The same, through one method or constructor the test calls.
`dispatch`	The test calls an interface or abstract member the target implements; a test double may have run instead.
`dispatch-via`	The same, through one method or constructor the test calls.

Tier	Meaning
`name`	The weak guess: the test's own name or its test class's name mentions the symbol at an identifier-segment boundary - for `Type.Member`, both names. A mid-word substring is not a mention.

The first six tiers are strong. The evidence reaches two hops at most: the test itself and one method or constructor it calls, or one property getter it reads. A deeper chain (helper calling helper) or a table-driven sweep is not counted, so an empty answer is not "nothing pins this". `invokesTotal` counts the strong rows and `dispatchTotal` the contract rows over the whole result, not only the returned page; `totalFound` minus both is the number of name guesses. Stronger evidence sorts first, so the cap can only drop the least load-bearing end.

Notes. The `note` says so on an empty answer, and names the other cases where the bare numbers mislead:

- A query that names nothing declared at all, or a qualified query whose type segment names no declared type. Matching is case-sensitive, so every hit is weak name noise - it describes the spelling, not this solution's coverage.
- An event or an enum member. Only the name tier can match it.
- A nested type in the `Outer.Inner` form.
- A constructor query (`Type.Type`).
- Contract rows (`dispatch`, `dispatch-via`), and construction rows, which are matched on the type's simple name, so a same-named type counts too.
- A non-empty answer in which no row is strong (`invokesTotal` 0) - only name guesses or contract rows. The note says that the list does not show that anything pins the symbol, and for a member it adds the declaring type's own count as a cross-check.

Example

```
{ "sessionId": "<session>", "symbol": "OrderService.Cancel" }
```

`coverage_gaps`

Purpose. Surface negative knowledge for a scope: methods that no test invokes directly, and semantic axes that are explicitly declared absent.

When to use it. Before claiming that code is covered; when you want a list of untested methods; when you want to know which semantic annotations are deliberately blank.

Availability. Default tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`scope`	string	no	`"solution"`	`"solution"` or a namespace prefix to narrow which symbols are checked.

Response. `scope`, an optional `note`, `symbolsScanned`, `uncovered` (a capped envelope, maximum 200), `verifiedAbsent` (a capped envelope, maximum 200), `transitivelyReached`, `judgedUnits`, `testProjectsExcluded`, `testFixtureTypesExcluded` and `reachedOnlyThroughDispatch`.

Field	Meaning
`uncovered`	Methods that no test invokes directly. Each item carries `kind`, `name`, `declaringType`, `namespace` and, when applicable, `testReachDepth`.
`verifiedAbsent`	Semantic axes the developer explicitly declared as `none`. Each item carries `kind`, `name`, `declaringType`, `namespace` and `field`.

Field	Meaning
`symbolsScanned`	Every unit in scope, methods and types.
`judgedUnits`	The non-test, non-abstract methods the uncovered list is drawn from - the population behind the gap count.
`testProjectsExcluded`	The distinct test-classified projects whose units were excluded.
`testFixtureTypesExcluded`	The distinct types excluded through the attribute-inferred fixture axis: a type that declares a test-attributed method is a test fixture, so its attribute-less `SetUp` /helper scaffolding is test code even when the project's name matches no test rule. Its constructors and its field and event initializers count as test code too.

The one-hop definition. The uncovered list is a one-hop index, not a coverage measurement. A method that a test reaches through another method - a CLI verb behind its command builder - is listed as uncovered although it is thoroughly exercised. Therefore read the per-entry `testReachDepth`: the hops to the nearest directly tested method, where 1 means a directly tested method calls it. Entries without that field are the ones no test reaches at all - work those first. A shallow depth is close to "already exercised" (confirm with `find_tests_for`); a deep one is more likely incidental.

The entries are marked rather than filtered out, because a deep transitive path is not the same claim as a test, and silently hiding a genuinely untested method would be the worse error for a tool whose job is negative knowledge.

Notes.

- A non-empty answer leads with a `note` carrying the split for that answer - how many of the listed methods are reached from a test through intermediate calls - and the same count is available as `transitivelyReached`. When no listed method is reached even indirectly, the note says so instead.
- A constructor is a hop of the walk, never an entry of the list: `new X(...)` reaches the constructor overload it binds and what that constructor calls, so a method only a constructor calls gets a `testReachDepth`. The walk also follows an object that a field or property initializer creates: the code its constructor calls counts as reached when a test creates the containing object. A call from an accessor is not read.
- The walk also follows interface and abstract dispatch; an entry reached only that way carries `viaDispatch`, and `reachedOnlyThroughDispatch` counts them.
- A `scope` that matches no namespace answers `symbolsScanned: 0` with a `note` that says where to look instead: the namespace of a type the prefix names, the namespaces that contain the prefix as whole segments, or the closest namespace.
- Abstract and interface method declarations are excluded: they have no body and are not coverable units.
- A weak test-name match does not count as coverage here; it is too weak to prove anything.
- Runtime coverage from a coverage tool remains the only measurement that sees reflection, generated and framework-invoked paths.
- Absence of a detected test is not proof that zero tests exist.

Example

```
{ "sessionId": "<session>", "scope": "MyApp.Application" }
```

`find_by_complexity_and_coverage`

Purpose. The prioritized refactor and test list: the methods that are both complex and have no direct test call - the intersection of the complexity ranking and the uncovered list, in one call, ranked by complexity descending so the riskiest float to the top.

When to use it. When you want to pick the next refactoring or test target by risk rather than by browsing two separate lists.

Availability. Not in the default tool set; reachable through a profile whose facet menu includes it (for example Refactoring, Debugging, Review, Testing or Performance), or via `AICB_MCP_TOOLS`.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`minComplexity`	int	no	`0`	Complexity floor. Only methods with cyclomatic complexity greater than or equal to this value are considered; 0 means no floor. The list is ranked by complexity descending regardless. Raise it, for example to 10, to keep only genuinely complex untested methods.
`scope`	string	no	`"solution"`	`"solution"` or a namespace prefix to narrow scope.

Response. `scope`, an optional leading `note`, `minComplexity`, `complexMethodCount`, `uncoveredMethodCount`, `methods` (a capped envelope, maximum 200, complexity descending) and `transitivelyReached`.

`complexMethodCount` and `uncoveredMethodCount` report the two input sizes - the complex methods at or above the floor, and the uncovered methods in scope. Each item carries `name`, `declaringType`, `namespace`, `cyclomaticComplexity`, `parameterCount` and, when applicable, `testReachDepth`.

Notes.

- The tool inherits the one-hop definition from `coverage_gaps`: a method a test reaches through another method is in this list too. Read the per-item `testReachDepth` and treat the items without it as the real candidates - the distinction matters more here than in `coverage_gaps`, because this output is a worklist people act on. A non-empty answer leads with a note carrying that split.
- There is no `includeTests` switch, deliberately: the complexity half excludes test-project methods, and a test method is by definition never untested.
- Overloads remain separate entries, each with its own complexity and parameter count; the shared `(namespace, declaring type, name)` key is only used to look up coverage.
- It composes two persisted-fact services and works on recalled sessions.

Example

```
{ "sessionId": "<session>", "minComplexity": 10, "scope": "MyApp.Core" }
```

`assert_absence`

Purpose. Check a negative claim against the analyzed model and return `confirmed`, `refuted` or `indeterminate` with the evidence.

When to use it. To validate statements such as "this service is untested" or "this axis is explicitly unset" before relying on them.

Availability. Default tool set.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`claim`	string	yes	-	One of the three claim forms below.

Claim	Meaning
`no_tests:<Symbol>`	The symbol has no detected covering test.
`absent:<Symbol>.<axis>`	The semantic axis is verified-absent - a developer explicitly declared `none`, not merely unknown.

Claim	Meaning
`unknown:<Symbol>.<axis>`	The axis has no provenance entry at all.

`<Symbol>` is a bare name or the qualified `Type.Member` form - the same strings that `find_usages`, `find_tests_for` and `impact_of_change` resolve. `axis` is one of the semantic axes, for example `role`, `layer`, `domain`, `context` or `responsibility`.

Response. `claim`, `claimType`, `symbol`, `axis`, `verdict`, `reason` and `evidence` (a list of strings).

Verdicts.

Verdict	Meaning
`confirmed`	The model supports the claim. For `no_tests` this means no covering test was detected. For `absent` the axis really is verified-absent. For `unknown` the axis really has no provenance entry.
`refuted`	The model contradicts the claim. For `no_tests` this requires at least one strong covering test, as `find_tests_for` defines its tiers. For `absent` the axis has a provenance source other than verified-absent.
`indeterminate`	The model cannot decide.

Notes.

- The tool is conservative in both directions: what the model cannot prove is `indeterminate`, never falsely confirmed - and never falsely refuted either.
- Tests that merely carry the symbol in their name yield `indeterminate` with the invokes/name-only split disclosed in the reason, not a refutation; a test named after a symbol is not proof that it exercises it.
- For a property, field, event or enum member, `no_tests` can only ever rest on name matches, because that member kind carries no invocation edge; the reason says so.
- A name that resolves to nothing is reported as such - a statement about the name, not about the code. A `no_tests` claim that names a declared type by its namespace-qualified name is not reported as undeclared: the reason names the type and asks for its simple name.
- An axis name that is not recorded anywhere in the solution returns `indeterminate` with the actually recorded axes as evidence. A typo or an invented axis is never silently `confirmed`.

Example

```
{ "sessionId": "<session>", "claim": "no_tests:OrderService.Cancel" }
```

7.5 Dependency injection

`resolve_injection`

Purpose. Resolve a `Microsoft.Extensions.DependencyInjection` or MEF registration: given a service type - an interface, or a class registered as itself - return what is registered for it, the lifetime, and where the registration happens - `location` (the registering call as file and line), `project` and `declaringMethod`.

When to use it. When you need to know which implementation is wired for an interface, which lifetime applies, or where a registration lives; when a fan-in or instantiation answer suggests a container registration. It does not answer usage (`find_usages`), implementers (`find_implementations`) or construction (`instantiation_sites`).

Availability. Default tool set. Note: this tool needs a live analyzed session; a session restored from a saved snapshot cannot answer it.

Parameters

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Session id, or an absolute `.sln` path for self-init.
`interface`	string	yes	-	The service type's simple or fully qualified name, for example `ICodeAnalyzer`. A closed generic form (`IHandler<Saved>`) lists only the registrations serving that closing, together with the open registrations (`IHandler<>`) that serve every closing; `IHandler` or `IHandler<T>` lists every closing. Aliases: `symbol`, `interfaceName`.
`site`	string	no	`null`	Narrow to registrations whose file path contains this substring.
`includeTests`	bool	no	`false`	Include registrations declared in test projects.

What is scanned. `AddSingleton`, `AddScoped` and `AddTransient`, their `TryAdd*` variants and keyed forms - in generic, `typeof`, instance and factory-lambda form. Beyond these:

- **Project-local registration helpers.** A helper declared in the solution that registers its own type arguments is followed into its body, whatever its name - a wrapper such as `AddCachedSingleton` as well as an `AddHandler<TEvent, THandler>()`. Each call of the helper is read with its own type arguments, and such a row carries the note `via <helper>`. A generic the helper builds over its type parameter is closed with the call's arguments; a keyed call inside the helper takes each call's own key (a key the helper computes from what it is handed reads `(dynamic)`); a factory lambda names its product where the body shows one type. Where a call read through the helper is in view, the helper's own body line is not listed again, and two identical `TryAdd`s that one call reaches under the same key are one row.
- **Service descriptors built at the call.** `Add`, `TryAdd`, `TryAddEnumerable` and `Replace` on the service collection, handed a `ServiceDescriptor` the call shows being built: in place (`new ServiceDescriptor(...)`, `ServiceDescriptor.Singleton` / `Scoped` / `Transient` / `Describe` and the keyed factories), in a local that is not written again, or as the elements of the collection a `TryAddEnumerable` takes (one row per element). A row from `TryAdd`, `TryAddEnumerable` or `Replace` is noted with that name; a plain `Add` carries no such note.
- `AddHostedService<T>()` and its factory overload: `IHostedService` implemented by `T`, Singleton, note `AddHostedService`.
- **Package helpers with an `As` chain.** A package extension such as `AddSingletonAs<T>()`, whose body the analysis does not read, followed by `.As<I>()` or `.AsSelf()`, is read from its call shape alone: it registers `I` (or `T` for `AsSelf()`), implemented by `T`, with the lifetime its name says. Such a row carries the note `inferred, not read: ...` naming the helper and its assembly, and the answer says once which helpers were read this way and that what else they register - and whether they add, try to add or replace - is not known.
- **One data-driven shape:** a `foreach` over a static table of `new(typeof(Service), typeof(Impl))` rows is expanded into one registration per row, also when the table lives in another project. Such a registration carries the note `table row <file>:<line>`.
- **MEF exports.** An `[Export]` attribute of `System.ComponentModel.Composition` or `System.Composition` (or an attribute derived from their `ExportAttribute`) on a class, struct or record, or on its methods, properties and fields, is a registration row. The service is the `typeof` argument, else the part or the member's type; `key` is a contract name as written; `implementationType` is the part, or `(factory)` for a member export. The lifetime: MEF 1 `[PartCreationPolicy(CreationPolicy.Shared)]` is `Singleton` and `NonShared` is `Transient`, otherwise `(dynamic)`; MEF 2 `[Shared]` on any partial declaration is `Singleton`, else `Transient`; an export whose attribute does not bind (an unrestored project) reads `(dynamic)`. The `note` reads `MEF export`, followed for a derived attribute by `via [Name]: the service is read from its type argument; the contract it hands MEF is set in its constructor` and for a member export by `; member Part.Member`. MEF rows never make `ambiguous` true and never win `effectiveRegistration`; beside Microsoft DI registrations the winner is judged over those alone.

What is not read: a `ServiceDescriptor` that is not built at the call or in a local there (a parameter, a loop variable, a method result, a conditional, a reassigned local, a collection held in a local, a spread element), framework helpers that register on the caller's behalf (`AddHttpClient`, `AddDbContext` and the like), Scrutor assembly scans, Autofac, Castle Windsor and other non-Microsoft containers, and of MEF: a derived export attribute without a `typeof`, a method export without a `typeof`, `[InheritedExport]` on an interface, a convention builder, and an `Export` attribute that does not bind in a file importing no MEF namespace (such as Xamarin's selector attribute). Umbraco collection builders are read, but as operations under `collectionBuilders`, not as registrations.

A call whose service type is a runtime value - reflection or assembly scans, a method result, a deconstructed loop variable - is not expanded, and neither is a factory lambda whose product cannot be named. Instead a `note` names those sites and which kind each one is. A site that demonstrably cannot register the queried service - for example one that registers a class as itself while the query is an interface - is left out of that list and only counted with its reason. An empty list with such a note means "not statically visible", not "not registered".

A factory that *returns* its object is resolved: both `sp => Foo.Build()` and a block body whose `return` statements all agree on one type. Where they disagree, or where there is no `return` at all, the product stays unnamed and the site is disclosed rather than guessed at. A `return` inside a nested lambda or local function belongs to that function, not to the factory.

Response. `query`, `matchCount`, `ambiguous`, `multiRegistration`, `registrationSites`, `consumedAsCollection`, `testRegistrationsFiltered`, and `registrations`. On a closed query the answer adds `typeArgumentFilter`; for an Umbraco collection builder it adds `collectionBuilders` (see the table), and the answer's `note` says `A collection builder collects types for this name: see collectionBuilders.` Each registration carries `serviceType`, `implementationType`, `lifetime` (`Singleton`, `Scoped` or `Transient`; `(dynamic)` for a MEF part whose creation policy is not fixed or not readable), an optional `note`, `location` (file and line of the registering call; a relative path such as `..\...` for a project outside the solution folder), `project`, `declaringMethod` (`Type.Method`; null for top-level statements), `key` for a keyed registration (omitted otherwise) and `isTestProject` when the registration comes from a test project. `serviceType` carries its type arguments - `IDisplayDriver<ISite>`, `IHandler<>` for an open registration, `IHandler<TModel>` where a type parameter of the surrounding code fills it - while `implementationType` is a simple name, the service itself for a self or instance registration. A keyed call (`AddKeyedSingleton<IFoo, Foo>("k")`, also the non-generic `AddKeyedSingleton(typeof(IFoo), "k", typeof(Foo))`) keeps its key: two keys are two services, and so are two closings (`IRepo<A>`, `IRepo<B>`).

Rows noted `inferred, not read` count as rows - in `matchCount`, `multiRegistration`, `registrationSites` and the closed-query filter - but not as evidence about how the helper adds: they take no part in `ambiguous`, never name an `effectiveRegistration` winner, and never make an optional dependency read `yes`.

Field	Meaning
`registrationSites`	How many distinct registration sites matched.
`multiRegistration`	Whether more than one registration matched at all.
`consumedAsCollection`	True when the service is consumed as a collection somewhere - a `GetServices<T>()` or `GetServices(typeof(T))` call, or an `IEnumerable<T>`-family or `T[]` constructor parameter, optional and nullable ones included. Several registrations are then a deliberate multi-binding. Consumers declared in test projects count only under `includeTests`, exactly as the registration list does.
`ambiguous`	Judged within one registration site (one declaring method, or the file outside any method) and per service (per key and per closed service type): true only when that site registers more than one distinct, statically resolvable implementation and `consumedAsCollection` is false. Two rows that both add to the service's set (`TryAddEnumerable`, `AddHostedService`) are no conflict. Across sites it is deliberately not ambiguity: a solution with several hosts (GUI, CLI, MCP) registers the same service once per host, and those containers never meet. Use `registrationSites`, `multiRegistration` and the per-registration `declaringMethod` to judge the case this grain cannot see: two helper methods composed by the same host.
`effectiveRegistration`	Present when two or more registrations match: `winner` is the registration a single-service resolve receives - the last one kept, with `TryAdd*` keeping nothing once the service is registered - read only where the order is certain, within one straight-line registration method. Otherwise `winner` is `null` and `reason` says why (for example several sites, a call inside a branch or loop, two types of one simple name, a `TryAddEnumerable`, `Replace` or `AddHostedService` with its rule named, or a row inferred from a package helper's call shape). A descriptor `Add` / `TryAdd` takes part in the order like `AddSingleton` / `TryAddSingleton`. Keyed registrations are set aside, because a plain resolve never receives one. When a service has two or more MEF exports and no Microsoft DI registration, `reason` says that MEF composes by contract and no export wins.
`constructorConsumerCount`	How many types take the service as a plain constructor parameter - visibility, never a usage verdict.

Field	Meaning
`otherConsumers`	The consumers that take the service other than through a constructor parameter, one entry per form found: `locator` (a `GetService<T>()`, `GetRequiredService<T>()` or `GetService(typeof(T))` call or its keyed twin, whatever the receiver - `Ioc.Default`, an `IServiceProvider` - also through `?.`) and `inject-property` (a C# property with `[Inject]`, a Razor `@inject` directive, an `[Inject]` property in a `@code` block, in `.razor` and `.cshtml` files outside build output). Each entry carries `form`, `consumers` (distinct declaring types; a Razor file, and a call outside any type, count as their own consumer), `sites` (calls or properties; a Razor file is one site) and `examples` (up to three, as `File.cs:line`; a Razor example carries no line). Test projects count only with `includeTests`. `GetServices<T>` stays with `consumedAsCollection`. Omitted when there are none.
`optionalDependencies`	The constructors taking the service as an optional parameter, each judged `yes`, `no` or `unknown`. On a MEF export row, an optional constructor parameter of the part is never judged `yes` (it reads `unknown`), because MEF fills an importing constructor from its imports.
`typeArgumentFilter`	On a closed query only: `typeArguments`, `otherClosings` (registrations of the same name closed with other arguments, left out; `matchCount + otherClosings` is the answer to the bare name under the same filters), `unresolved` (rows whose closing the call does not fix, kept in the list) and a `note` saying which fields stay at the level of the bare name - `otherConsumers` among them, which reads every closing.
`testRegistrationsFiltered`	How many registrations the default test filter removed. Omitted when nothing was hidden.
`collectionBuilders`	Umbraco collection builders whose collection, item type, builder or collected types the query names - listed apart, never as registrations: `registrations`, `matchCount` and the other fields are unchanged by them. A `note`, and per builder `builder`, `collection`, `item`, `operationCount`, `unreadCount` and up to 100 `operations` in the order they are written (by project, file and position), each with `operation` (`append`, `insert`, `insertBefore`, `insertAfter`, `add`, `remove`, `replace`, `exclude` or `clear`), `type`, `anchor` (for `insertBefore`, `insertAfter` and `replace`), `location` and `project`. `type` is null for `clear`, for a type passed as a runtime value and for a generic helper's own type parameter; the last two count in `unreadCount`. The builder registers each collected type as itself and the collection through a factory, so nothing registers the item type, and the resulting order and set depend on the order the composers run in. A builder is recognized by Umbraco's `CollectionBuilderBase` (namespace `Umbraco.Cms.Core.Composing`); a namesake in another namespace is not. The `site` and test filters apply to the operations as to the list. Omitted otherwise.

The three empty answers are distinguishable.

1. A name nothing in the solution names - no declaration, no registration, no base list, parameter, type argument, `typeof`, Razor `@inject` or `[Inject]` property that writes it - carries `resolvedKind: "not_found"` and a `nearest` suggestion: a typo. A closed query whose name is registered with other closings is not `not_found`; its note names the closed form.
2. A real name with no registration always carries a `note`. Its basic form opens `No Microsoft.Extensions.DependencyInjection (Add*/TryAdd*, keyed included) registration and no MEF [Export] naming '<name>' was found` and states the reader's boundary - nothing was found, which means "not statically visible", never "not registered" - scoped to the answer's view (it says when registrations in test projects or outside the `site` filter were left out), and it names the forms that are not read together with which of them the solution actually holds: - registration helpers whose body is not read - a package's, or one a source generator writes - by call count, pointing first at helpers from the assembly that declares the queried type or named after it, and at calls that pass the type or an implementer as a type argument; - registration by convention: types marked `ITransientDependency`, `ISingletonDependency`, `IScopedDependency` or `[ExposeServices]` are counted, with a pointer to the types that name the service in their base list or in `[ExposeServices]` (a non-empty answer gets the pointer alone); - MEF parts exported through `[InheritedExport]` on an interface or a convention builder (an `[Export]` attribute is read); - calls whose service type is a runtime value, as described above; - for an interface the solution does not declare, that the library declaring it may register it in code this scanner does not see; - when a scanned project references a foreign container assembly, that sign, because its registrations are invisible here.

3. Everything was filtered out by the test filter, reported as `testRegistrationsFiltered: N`; the note says how many were left out and that `includeTests: true` lists them.

Notes.

- Test-project registrations are excluded by default - a test's own container stub is not the production binding. Set `includeTests: true` to include them; each one is flagged `isTestProject`. The same filter reaches collection CONSUMPTION: a fixture taking `IEnumerable<T>` does not, on its own, mark a production conflict between two registrations as deliberate.
- A service consumed as a collection is not flagged as ambiguous even with several registrations; see `consumedAsCollection`.

Example

```
{ "sessionId": "<session>", "interface": "ICodeAnalyzer", "site": "DependencyInjection" }
```

8 Tool reference: facts, dead code, metrics and patterns

This chapter covers three groups of tools that ask about the analyzed model itself rather than about one symbol's neighborhood:

- deterministic fact queries - side effects, external calls, concurrency risks, resource leaks, event subscriptions, attributes, return roles, code traits, resolved semantics and provenance;
- dead code, namespace cycles and metrics;
- structural patterns and documentation drift.

All nineteen tools are read-only: none of them changes a file, a database or the session, and none of them needs a running GUI. Unless a tool says otherwise, it reads facts that were stored during the analysis, so it also works on a session you recalled from the database. All of them can be dispatched through `batch` and `measure`, as long as the active tool set exposes them.

Shared conventions

- `sessionId` accepts either the session id returned by `analyze_solution` or the absolute path of a `.sln`, `.slnx` or `.slnf` file. With a path, the server analyzes the solution on first use.
- `scope` is `"solution"` by default. Any other value is a namespace prefix, matched case-insensitively at a segment boundary: `MyApp.Core` matches `MyApp.Core` and `MyApp.Core.Services`, but not the sibling namespace `MyApp.CoreX`. A prefix that is too deep or misspelled simply matches nothing, and the answer then says so.
- `includeTests` is `false` by default, so test-project units are excluded. Most responses report removals in their own field (`testMatchesFiltered`, `testMethodsFiltered` or `testTypesFiltered`). `detect_circular_dependencies` is the exception: it has no filtered-count field, so cycles that disappear with the test graph are not disclosed separately.
- Capped lists come as an envelope `{ items, count, totalFound, truncated }`: `count` is the length of the delivered list, `totalFound` the true total.
- A leading `note` qualifies an answer that would otherwise be misread - above all a zero. Read it before acting on the numbers.
- Where a `kinds` parameter exists (`find_by_attribute`, `find_dead_code`, `find_production_dead`), a value other than the documented ones is treated as the full superset - never as "nothing".
- The fact queries echo the filter you passed in the response; when you passed none, the field reads `(any)`.
- Committed machine-written files (`*.Designer.cs`, `*.g.cs`) are read for references, but what they declare is never judged: `find_dead_code`, `find_production_dead`, `find_god_objects`, `detect_circular_dependencies` and `check_doc_drift` - like `coverage_gaps`, `find_by_complexity_and_coverage` and the insights - do not report it. The fact queries still list such methods.

Availability. Nine of the nineteen tools in this chapter belong to the default tool set. The other ten are marked "not part of the default tool set" and need the `Full Select` profile or an `AICB_MCP_TOOLS` override:

```
aicb mcp --mcp-profile mcp-profile/full
```

Alternatively, expose them with the `AICB_MCP_TOOLS` environment variable (`all`, or a comma-separated list of tool-class names) before starting the server.

8.1 Facts about side effects, external calls, attributes and semantics

Every tool in this group reads stored Roslyn facts, so it works on a recalled session as well as on a live one. `confidence_map` and `find_by_semantics` read the resolved semantic fields; the others read the method and type facts produced by the analysis.

`find_by_side_effects`

Finds methods by their observable side-effect profile - what they touch in the outside world. Use it to answer "what goes to the database or the network?" and "is this method pure?".

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`effect`	string	no	-	Filter to exactly one effect token (case-insensitive). The vocabulary is closed: any other value is rejected with an error that names the token you sent and the eight valid ones.
`purity`	string	no	`any`	`any`, `pure` (no detected side effect - a good refactoring and test-isolation signal) or `impure`. Any other value is rejected.
`scope`	string	no	`solution`	`solution` or a namespace prefix.
`includeTests`	boolean	no	`false`	Include test-project methods.

The eight effect tokens:

Token	What it asserts
`io`	Filesystem and stream access. A `System.IO` type counts on contact, not on topic: path strings (`Path`), glob matching (`FileSystemName`) and the in-memory streams and readers (`MemoryStream`, `StringReader`, `StringWriter`) carry no effect, while `File`, `Directory`, `FileStream`, `FileInfo`, `StreamReader` and `StreamWriter` do.
`network`	HTTP, socket, mail and cloud-storage calls. A `System.Net` type counts on contact, not on topic: addresses, cookie and header containers, credential holders, the HTTP and mail message surfaces and all of `System.Net.Mime` carry no effect, while `Dns`, `Sockets`, `WebClient`, `SmtpClient`, `HttpClient`, `HttpContent` and similar clients do.
`database`	Database drivers and ORMs, for example ADO.NET and `System.Data`, EF Core, Dapper, `Microsoft.Data.Sqlite`, Npgsql, MongoDB and Cosmos.
`serialization`	(De)serializer calls, for example `System.Text.Json`, Newtonsoft.Json, `XmlSerializer`, data contracts, MessagePack, protobuf and YamlDotNet.
`logging`	Logger calls, for example `Microsoft.Extensions.Logging`, Serilog, NLog and log4net.
`cache`	Calls into an external cache API, for example `Microsoft.Extensions.Caching`, StackExchange.Redis, EasyCaching or FusionCache. A cache a type holds in memory is not an outside-world effect and is never classified here.
`messaging`	Message-bus and queue clients, for example MassTransit, RabbitMQ, Azure Service Bus/Event Grid/Queues, Kafka, NATS, MediatR and Amazon SQS/SNS.
`unknown`	An external call that no classification rule matched - an unanalyzable third-party member, reported so the gap is visible instead of reading as pure.

Response. `scope`, `note`, `effect`, `purity`, `methodsScanned`, `methodsScannedNote` (on `purity="pure"` only), `matches`, `availableEffects` and `testMatchesFiltered`. Each hit carries the method's `name`, `declaringType`, `namespace` and its `effects` list.

Notes.

- Effects are propagated transitively, along calls and creations: a method that calls a project method which hits the database counts as `database`, and so does a method that builds an object whose constructor hits the database. A constructor that chains to such a constructor with `: this(...)` or `: base(...)` carries the effect too. A hit does not say which of its effects are direct and which inherited.
- With neither `effect` nor `purity`, only impure methods are listed - a bare "all methods" dump has no value.
- The scan set is every method plus every property, indexer and event accessor and every constructor with a body, each as a unit of its own under its metadata name (`get_X`, `set_X`, `.ctor`, `.cctor`). A field or property initializer counts toward the constructor the compiler runs it in. The `purity="pure"` list stays methods-only, because a bodied getter is pure by default and would drown it. A pure answer says so in `methodsScannedNote`: its `methodsScanned` is smaller than that of the impure and unfiltered listings by design, so compare listings by their hits, not by this count.

- **Top-level statements.** On an unscoped query each file of top-level statements is one unit, listed as `<path>/Program.cs (top-level)` with no declaring type and no namespace, beside the methods; `methodsScanned` counts it once. Its effects include what the project methods and constructors it calls reach, and a local function among the statements is part of it. A namespace-scoped query and the `purity="pure"` list stay methods-only; a test project's file is filtered like a test method, and a multi-targeted project's file counts once. On a capped page the top-level rows sort first; `totalFound` and `truncated` say what was cut. On a session recalled from a snapshot stored before these units were recorded, the answer carries `TOP-LEVEL EFFECTS NOT KNOWN`.
- On zero matches, `availableEffects` lists the tokens this scope actually records. An empty `availableEffects` means nothing was recorded in a scope that was scanned; the field is omitted when `methodsScanned` is `0`, and the note then says that nothing was scanned at all.
- `effect` together with `purity="pure"` is rejected: a pure method carries no effect, so the answer would be empty by construction.
- If the analysis run could not resolve some projects' references, a leading note warns that effects propagate along call edges, so methods can read as pure that are not. Cross-check with `get_diagnostics`.
- The list is capped at 200.

`calls_external`

Finds the production methods that call an external (non-solution: BCL or third-party) method whose fully qualified key contains a pattern - the "who calls this external or risky API?" question that `find_usages` cannot answer, because it stays inside the solution. Typical uses: AOT and trimming readiness, security surface, external dependency footprint.

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`pattern`	string	yes	-	A case-insensitive substring of the external call key, for example `Process.Start`, `File.`, `Assembly.Load`, `System.Reflection` or `JsonSerializer`. `apiName` is accepted as an alias. A very broad pattern such as `System` is truncated at 200. It is the only search argument; there is no `symbol` parameter.
`scope`	string	no	`solution`	`solution` or a namespace prefix.
`includeTests`	boolean	no	`false`	Include test-project methods.

Response. `scope`, `note`, `pattern`, `methodsScanned`, `matches` and `testMatchesFiltered`. A hit carries `project` only where two projects compile the same type, so two such rows stay apart. Each hit reports the two kinds apart: `matchedCalls` for invocations and `matchedReads` for external property reads. An invocation executes, while a read only observes ambient state, and an AOT, testability or security verdict differs per kind.

Notes.

- Two stored facts back the tool: external method invocations (keys such as `System.IO.File.ReadAllText(string)`) and external property reads (such as `System.DateTime.UtcNow`). The `global::` prefix is stripped before matching and display.
- The scan set includes every property, indexer and event accessor and every constructor with a body as a unit of its own; a member initializer counts toward the constructor that runs it. An expression-bodied getter `=> DateTime.UtcNow` is therefore a read site under its own name.
- On an unscoped query each file of top-level statements is one unit as well, listed as `<path>/Program.cs (top-level)` with no declaring type and no namespace, with its calls and reads split as for a method and a local function among the statements included; `methodsScanned` counts it once. A namespace-scoped query stays methods-only. On a session recalled from a snapshot stored before these units were recorded, the answer carries `TOP-LEVEL EFFECTS NOT KNOWN`.
- The pattern is open vocabulary: an empty result is not a misspelling and there is no suggestion field. Empty is a strong statement but not an absolute one - an external field read (`string.Empty` and other value carriers, deliberately excluded as audit-irrelevant) and a bare `new X()` are not recorded either.

- An empty answer that is only the test filter's doing says so instead of claiming nothing matched, and points you to `includeTests=true`. `testMatchesFiltered` tells you how many matching methods were dropped.
- A leading note appears on any answer, empty or not, when the analysis run failed to resolve some projects' references: an unresolved call records no external key at all, so the index is short by ordinary calls.
- The list is capped at 200.

`find_by_concurrency_risk`

Finds methods with a concurrency or async-safety smell - a deterministic Roslyn anti-pattern that no other tool surfaces.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`risk`	string	no	-	Filter to one token: `sync-over-async` (blocking on a task via `.Result`, `.Wait()` or `.GetAwaiter().GetResult()` - a deadlock risk), `async-void` (an `async void` method that is not an event handler - its exceptions cannot be caught and it cannot be awaited) or `fire-and-forget` (a task-returning call left as a bare statement, its result dropped - unobserved exceptions, lost work). Omit to list every method with any concurrency smell.
`scope`	string	no	`solution`	`solution` or a namespace prefix.
`includeTests`	boolean	no	`false`	Include test-project methods.

Response. `scope`, `note`, `risk`, `methodsScanned`, `matches`, `availableRisks` and `testMatchesFiltered`. Each hit carries the method's `name`, `declaringType`, `namespace` and its `risks` list.

Notes.

- The detection is conservative and semantic: an own `Result` property on a non-task, an `async void` event handler, and an awaited, assigned or returned task are not flagged. Accessors and constructors with a body are scanned as units of their own, so a `sync-over-async` `.Result` inside a getter is found.
- The list is inherently small - most methods carry no smell. Omit `risk` to see all of them.
- Read `testMatchesFiltered` before concluding that a token is wrong: a filtered zero arrives next to an `availableRisks` list that does not contain the queried token, which reads like a misspelled token although the token was right and only its carriers were test code. A blocking `.Result` deadlocks a test run exactly as it does production code.
- On zero matches, `availableRisks` lists the tokens present in scope; an empty list means none was recorded in a scope that was scanned (the field is omitted when nothing matched the scope).
- The list is capped at 200.

`find_by_resource_leak`

Finds methods that leak a disposable - an `IDisposable` or `IAsyncDisposable` created with `new` whose ownership is neither transferred nor released, such as an unclosed file, socket or database handle.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`disposableType`	string	no	-	Filter to methods leaking this type (case-insensitive simple name, for example `FileStream` or `SqlConnection`). Omit to list methods leaking any disposable.
`scope`	string	no	`solution`	`solution` or a namespace prefix.

Parameter	Type	Required	Default	Meaning
`includeTests`	boolean	no	`false`	Include test-project methods.

Response. `scope`, `note`, `disposableType`, `methodsScanned`, `disposableCreationsChecked`, `disposableFactoryReturnsNotChecked`, `factoryReturnTypes`, `matches`, `availableDisposableTypes` and `testMatchesFiltered`. Each hit carries the method's `name`, `declaringType`, `namespace` and its `leakedTypes`.

Notes.

- Only the two unambiguous drops are reported: a bare `new X();` statement, whose result is immediately discarded, and `var x = new X();` where the local is never referenced again. Everything that transfers or keeps ownership is not flagged: a `using` statement or declaration, `return new X()`, a field or property assignment, a constructor or method argument (`new StreamReader(stream)`), a fluent `new X().Do()`, or a local that is used or disposed later.
- Constructors and accessors with a body are scanned as units of their own, so a constructor that drops a `new FileStream(...)` is found.
- `disposableCreationsChecked` is the denominator of the whole answer: the disposable creations (leaked or not) the two rules were applied to. A zero therefore reads as "0 leaks among N creations checked". **0 of 0** means nothing was looked at - either no creation in scope, or the run could not resolve the created types, because disposable-ness is a semantic verdict and an unresolved type counts in neither number. Check `get_diagnostics` for the scope.
- `disposableFactoryReturnsNotChecked` with `factoryReturnTypes` counts and names the disposables the same units obtained without `new` (a factory call or an awaited result; tasks are excluded). They are counted but not judged, because a factory does not always hand over ownership. A dropped `MSBuildWorkspace.Create()` or `Process.Start()` hides there; read those units by hand.
- On a zero with a `disposableType` filter, `availableDisposableTypes` lists the leaked types actually present in scope - read that empty filtered result as "no such leaked type in production", not as "no leaks", and use `testMatchesFiltered` to see whether the filter is the reason. On an unfiltered zero the list is empty by construction, because every leaked type would have matched; there the denominator carries the information.
- If the analysis run could not resolve some projects' references, a leading note says that both the leak list and the denominator are short by ordinary creations, and that a zero should be treated as unmeasured. A green `get_diagnostics.incompleteProjects` does not settle this: that check only asks whether core references bind, while facts degrade per assembly.
- The list is capped at 200.

`find_by_event_subscription`

Finds the types that subscribe to a C# event - a `+=` whose left-hand side is an event - anywhere in the type: constructor, method, property accessor or lambda. Event handlers are most often wired in a constructor, and this whole-type walk captures that pattern, which a per-method view misses.

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`eventName`	string	no	-	Filter to types subscribing to this event. Matched by the event name (`Click`) or the qualified key `DeclaringType.EventName` (`Button.Click`), case-insensitively. Omit to list every type with any subscription.
`scope`	string	no	`solution`	`solution` or a namespace prefix.
`includeTests`	boolean	no	`false`	Include test-project types.

Response. `scope`, `note`, `event`, `typesScanned`, `subscriptionSitesChecked`, `subscriptionSitesUnresolved`, `matches`, `availableEvents` and `testMatchesFiltered`. Each hit carries the subscribing type's `name`, `namespace` and the `events` it subscribes to. A partial type is one row with the events of all its declarations; `files` lists the subscribing files when there are

several. Rows that would read alike add `fullName` (two types of one name, such as `Foo` and `Foo<T>`) or `project` (the same type compiled in two projects, as with a linked file). `typesScanned` counts types, a partial type once.

Notes.

- Only a resolved event left-hand side counts. A `+=` on an `int`, a `string` or a delegate field (delegate combination, not an event) is skipped. This version captures subscriptions (`+=`) only.
- `subscriptionSitesChecked` is the number of `+=` sites the rule was applied to across the scanned units, and `subscriptionSitesUnresolved` how many of them had a left-hand side whose type did not resolve and were therefore not judged at all. The pair separates three different zeros: `0 of 0` (nothing to see), `0 of 7, 0 unresolved` (seven `+=` on non-events - a real absence) and `0 of 7, 7 unresolved` (an unmeasured scope). The unresolved-site case is flagged by the pair itself. A named event whose scope records no subscription at all also carries a did-you-mean note, and a scope with nothing scanned carries a separate "nothing scanned" note.
- The unresolved count is `0` on any solution whose types resolve, and it is not implied by `incompleteProjects`: that check only asks whether core references bind, so a project that resolves `System.Object` and fails on one framework or NuGet assembly loses every fact typed by it while still looking healthy. Use `get_diagnostics` to name the projects, then restore or rebuild, refresh the session and run again.
- Read `testMatchesFiltered` first when a named event comes back empty: a filtered zero arrives beside an `availableEvents` list that does not contain the queried name, which reads like a wrong name although the name was right and only its subscribers were tests.
- On zero matches, `availableEvents` lists the subscribed event keys present in scope. An empty list means no subscription was recorded in a scope that was scanned - read that as unmeasured, not as an absence, because recognition needs a resolved left-hand side.
- The list is capped at 200.

`find_by_attribute`

Finds symbols that carry a given attribute - the cross-cutting query that a name search cannot do. Use it for `[Obsolete]`, `[ApiController]`, `[Authorize]`, test attributes and similar.

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`attribute`	string	yes	-	The attribute to find, for example `Obsolete`, `ApiController` or `Fact`. `attributeName` is accepted as an alias.
`kinds`	string	no	`all`	`all` (types + methods + properties + fields), `type`, `method`, `property` or `field`.
`scope`	string	no	`solution`	`solution` or a namespace prefix.

Response. `scope`, `attribute`, `kinds`, `symbolsScanned` and `matches`. Each hit carries its `kind`, `name`, `declaringType`, `namespace` and the raw stored `attributes` list, so you see the ground truth. On the type axis a partial type is one row (its attribute lists combined) and is counted once in `symbolsScanned`; type rows that would read alike add `fullName` (`Foo` and `Foo<T>`) or `project` (the twin rows of a linked file).

Notes.

- The attribute name is matched format-independently: the last name segment, with an optional `Attribute` suffix stripped, case-insensitively. `Obsolete`, `ObsoleteAttribute` and `System.ObsoleteAttribute` all match the same symbols. This reconciles the two storage formats - types store the fully qualified attribute class, methods, properties and fields store the source syntax.

- The `method` axis includes constructors and accessors with a body (named `.ctor`, `get_X`, `set_X`), so a `[JsonConstructor]` or an `[Obsolete]` setter is found. Member scanning surfaces a member-level attribute such as a CommunityToolkit `[ObservableProperty]` on a backing field or a partial property, or a `[JsonProperty]`, which a type- and method-only scan misses.
- This tool has no test filter: it scans the whole scope, test projects included.
- An unrecognized `kinds` value behaves like `all`.
- The list is capped at 200.

`find_by_returns_semantic`

Finds methods by their inferred return role - what kind of operation the method is, derived deterministically from its signature. Useful for naming, decomposition and test-pattern reasoning: "list all the predicates", "which methods are factories?".

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`returnsSemantic`	string	yes	-	The role to match: `command`, `query`, `predicate`, `factory`, `mapper`, `validator`, `initializer`, `handler` or `builder`. Exact match, case-insensitive.
`scope`	string	no	`solution`	`solution` or a namespace prefix.
`includeTests`	boolean	no	`false`	Include test-project methods.

Response. `scope`, `note`, `returnsSemantic`, `methodsScanned`, `matches`, `availableReturnsSemantics` and `testMatchesFiltered`. Each hit carries the method's `name`, `declaringType`, `namespace` and its role.

Notes.

- The role is computed from the method name and return type by an ordered rule set - the first matching convention wins. A method that matches no convention carries no role and is not listed.
- This is a deterministic Roslyn fact, distinct from `find_by_semantics`, which queries the resolved role that may be asserted or heuristically guessed.
- The test filter is the reason this tool exists in its current form: `Validate*` and `Assert*` test methods would otherwise own the role list. `testMatchesFiltered` shows how many were removed.
- On zero matches, `availableReturnsSemantics` lists the roles actually present in scope, so read an empty result as "unknown role", not as "no such methods in production"; `testMatchesFiltered` tells the two apart. An empty `availableReturnsSemantics` means none was recorded in a scope that was scanned.
- The list is capped at 200.

`find_by_code_traits`

Finds methods by a deterministic code trait - what a method's body does, computed by Roslyn and stored. Use it for hot-path and allocation reviews, AOT and trimming audits, and error-path reviews.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`trait`	string	yes	-	One of `linq`, `linq-in-loop`, `reflection`, `throws`. Case-insensitive; any other value is rejected.
`scope`	string	no	`solution`	`solution` or a namespace prefix.

Parameter	Type	Required	Default	Meaning
`includeTests`	boolean	no	`false`	Include test-project methods.

What each trait means:

Trait	Meaning
`linq`	Uses LINQ operators - relevant for hot-path and allocation review.
`linq-in-loop`	A real `System.Linq` operator evaluated inside a loop body - a per-iteration allocation and complexity smell. This is a strict subset of `linq`.
`reflection`	An external call or read whose receiver type is reflection infrastructure: `System.Reflection.*`, `Type.*`, `Activator`, `AppDomain`, `object.GetType()` or expression trees. Matched on the receiver, never on a bare member name, so a WPF `DependencyObject.GetValue` / `SetValue` accessor or a `typeof(...)` in an attribute or dependency-property registration does not count.
`throws`	Contains a `throw` statement - relevant for error-path review.

Response. `scope`, `note`, `trait`, `traitReading` (on `reflection` only), `methodsScanned`, `matches` and `testMatchesFiltered`. Each hit carries the method's `name`, `declaringType` and `namespace`. A `reflection` answer carries `traitReading`: this trait is anchored on the receiver of a call or read, while the `UsesReflection` flag in the exported `METHOD_INFO` is set by other signals (any `typeof`, bare reflection member names), so the two can disagree about the same method in either direction. Both readings are kept on purpose.

Notes.

- Two traits are deliberately not offered: `isAsync` is already visible in the signature, and I/O is covered by `find_by_side_effects` with the `io` token.
- This axis has no suggestion list, so `testMatchesFiltered` is the only thing that separates "no such methods in the scanned set" from "no such methods". Check it before concluding anything.
- `reflection` and `linq-in-loop` exist only where a call target resolved. In a project with unresolved references they are false for every method there. Such a run leads the answer with a note naming the unresolved projects - cross-check with `get_diagnostics.incompleteProjects`.
- Accessors and constructors with a body are scanned as units of their own (`get_X`, `set_X`, `.ctor`).
- The list is capped at 200.

`find_by_semantics`

Finds symbols whose resolved semantics match the given criteria. All provided filters must match (logical AND). Use it to find, for example, all domain services, all view models, or everything whose responsibility mentions a keyword.

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`layer`	string	no	-	Match the resolved layer exactly (whole value, case-insensitive): `Domain`, `Application`, `Infrastructure`, `Presentation` or `Contracts`.
`role`	string	no	-	Match the resolved role exactly. Inferred type roles: `Service`, `Helper`, `Model`, `Record`, `Interface`, `Enum`, `Builder`, `ViewModel`, `Converter`, `Controller`, `Middleware`, `Attribute`, `Exception`, `EntryPoint`, `ContractModel`, `Type`. There is no `Repository` role - repositories classify as `Service`. Methods carry a method-category role, for example `Query`, `Command`, `Mapping`, `Validation` or `Orchestration`.

Parameter	Type	Required	Default	Meaning
`domain`	string	no	-	Free text, matched as a case-insensitive substring of the resolved domain (derived from naming and namespace).
`responsibility`	string	no	-	Free text, matched as a substring. Types only - methods have no such axis.
`minConfidence`	string	no	`any`	`any` (includes guessed values) or `asserted` (only `fact` and `ai`). Any other value is rejected.

At least one filter is required.

Response. `criteria`, `note`, `minConfidence`, `matches`, `availableAxisValues` and `nearMisses`.

- `criteria` echoes the applied filters and their mode: `role=View` for an exact closed-vocabulary match, `domain~order` for a substring match.
- Each hit carries the symbol's `kind`, `name`, `declaringType`, `namespace` and, per matched axis, the matched value together with its provenance: `fact`, `ai` or `inferred`.
- `nearMisses` appears whenever a closed axis (`role`, `layer`) was queried and some present value contains the query without being it. Each entry names the value and how many symbols carry it. These values are counted apart and named in a leading note, never listed as matches.

Notes.

- The difference between exact and substring matching is the point of this tool: `role="View"` is one view, not every view model.
- On zero matches, `availableAxisValues` lists the values actually present in this solution for each queried axis - read an empty result as "unknown value", not as "no such symbols".
- Asserted values can differ from the inferred defaults and keep their own spelling, for example `viewmodel` next to the inferred `ViewModel`. The case-insensitive match folds `viewmodel` into `ViewModel`; hyphenated refinements stay apart and show up as near misses.
- The list is capped at 200.

`confidence_map`

Reports the provenance of each semantic field of a symbol or a scope: is the value a deterministic Roslyn truth, a developer assertion, a heuristic guess, or explicitly declared absent? Use it to know what is known and what is guessed before you trust a symbol's role, layer or domain.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`symbol`	string	no	-	An exact, case-sensitive type or method name for one symbol. Omit to map a whole scope.
`scope`	string	no	`solution`	When `symbol` is omitted: `solution` maps every symbol, or a namespace prefix narrows it. Ignored when `symbol` is given.
`summaryOnly`	boolean	no	`false`	Return only the solution-wide tally, without the per-symbol entries. Use it on a large solution where the full per-symbol map would exceed the token limit.

Response. `note`, `scope`, `entries` and `tally`. The scope echoes `symbol` when you queried one symbol.

- Each entry carries the symbol's `kind`, `name`, `declaringType`, `namespace`, its `fields` (each field with its provenance token) and the per-token counts `factCount`, `aiCount`, `inferredCount` and `verifiedAbsentCount`.

- The four provenance tokens are: `fact` (deterministic Roslyn truth), `ai` (developer-asserted via an `<ai>` annotation), `inferred` (heuristic guess, low confidence) and `verified-absent` (the developer explicitly declared "none").
- `tally` aggregates over the scanned scope: the four token counts (summed over fields), `symbolsScanned` and `symbolsWithoutProvenance`.

Notes.

- The axes recorded per symbol include role, layer, context, domain and responsibility, among others; the map reports whatever is recorded.
- On a codebase with no `<ai>` annotations at all (fact, ai and verified-absent all zero - the normal state of a foreign codebase), a solution-scope answer leads with a note saying that every value is heuristically inferred, because the known-versus-guessed distinction does not exist there. A narrowed or per-symbol map is scope-filtered and cannot carry that codebase-wide claim.
- `summaryOnly=true` omits the per-symbol entries; the entries envelope still reports the true total and `truncated`.
- The entries are capped at 200.
- Works on a recalled session; provenance is stored.

8.2 Dead code, cycles and metrics

`find_dead_code`

Lists dead-code suspects as a queryable list, where the quality insights only give a top-20 aggregate.

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`scope`	string	no	`solution`	`solution` or a namespace prefix.
`includeTests`	boolean	no	`false`	Include test-project methods and types.
`kinds`	string	no	`method`	`method` - private methods with no detected caller; `type` - types with no incoming reference in the reliable reverse type fan-in; `all` - both.

Response. `scope`, `note`, `methodsScanned`, `deadMethods`, `typesScanned`, `deadTypes`, `methodsIneligible`, `typesIneligible`, `typesUnexamined`, `testMethodsFiltered`, `testTypesFiltered`, `methodsIneligibleNotPrivate`, `methodsIneligibleFrameworkInvoked`, `methodsIneligibleUnanalyzedCallers`, `typesIneligibleStructural`, `typesIneligibleTestScaffolding`, `typesIneligibleFrameworkActivated`, `typesIneligibleMarkupReferenced` and, where they apply, `generatedCodeNote` and `publicApiNote`. The counters are omitted at zero. Each dead method carries `name`, `declaringType` and `namespace`; each dead type carries `name`, `kind`, `accessibility` and `namespace`, and `publicApi: true` on a library's public type; a candidate in a project where a source generator did not run also carries `missingGeneratedCode: true`. Each axis is capped at 200.

Notes.

- The method axis is deliberately limited to private methods: public and internal methods have framework escapes - reflection, dependency injection, event handlers, source-generated callers - that the call graph cannot see. A private method a framework may call (`[RelayCommand]` and other framework attributes, partial methods, markup or designer event handlers, a static `Main`, a record's `PrintMembers`, `ShouldSerialize` / `Reset` methods) is skipped, and so is a method whose caller list was not analyzed (unresolved references, explicit interface implementations - also one whose interface did not bind, counted under `methodsIneligibleUnanalyzedCallers`) - skipped, not guessed. A markup-only caller counts as a real caller, so for example a Blazor `@onclick` handler is not reported, and a `nameof` of a method inside the attributes of a method (a `CanExecute` name) counts as a call.
- Committed `*.Designer.cs` and `*.g.cs` files are read for references but never judged: nothing declared in them is reported as a candidate.

- A partial type is one candidate and one scanned type, judged as a whole: an exemption one of its declarations earns (a static `Main`, for example) holds for the whole type.
- A candidate marked `missingGeneratedCode: true` sits in a project where a source generator or the WPF markup compiler did not run; `generatedCodeNote` counts them. Treat them as unjudged, not as dead - the missing code may be their caller.
- The type axis subtracts the structurally fan-in-invisible forms: interfaces, static and `*Extensions` classes, framework-, DI- and reflection-activated types (a MEF part - a type with an export attribute on itself or on a method, property or field - among them, counted under `typesIneligibleFrameworkActivated`), WPF/XAML types and MVC controllers.
- A type is used when a `.cshtml` view the Razor generator compiled, a file of top-level statements, or markup naming it as a nested `prefix:Outer+Inner` type references it (see `find_usages`). A generic type parameter named like a solution type is not a use of it.
- **Library public API.** A type of an assembly that builds a library (DLL or winmd) and is visible outside it - public, or protected or protected internal inside a derivable container, all the way out - stays listed and carries `publicApi: true`. A consumer outside this solution records no edge here, and the solution cannot say whether the library is published. `publicApiNote`, ahead of the lists and omitted when nothing is marked, counts the marked types (`LIBRARY PUBLIC API: <marked> of the <listed> listed types carry publicApi:true ...`) and says to settle whether the library ships before deleting one. Nothing is exempted, so an application built from libraries can list many marked rows.
- When `.cshtml` views were not compiled by the Razor generator, a non-empty candidate list carries the `VIEWS NOT READ` sentence (see `find_usages`).
- An answer that lists candidates leads with a note saying what the zero behind each candidate does not prove, and how to confirm one before deleting it.
- An empty answer is qualified only when the absence was structurally guaranteed. `methodsIneligible` counts scanned units the predicate could never judge and is split into `methodsIneligibleNotPrivate` (the deliberate policy), `methodsIneligibleFrameworkInvoked` (private methods a framework may call, as listed above) and `methodsIneligibleUnanalyzedCallers` (private methods with no analyzed caller list - the blind spot this axis exists to find). `typesIneligible` is split into four exemption families: `typesIneligibleStructural`, `typesIneligibleTestScaffolding`, `typesIneligibleFrameworkActivated` and `typesIneligibleMarkupReferenced`.
- If none of the scanned units was eligible - an abstractions namespace consists only of interfaces, for example - the note says that the scope could not have produced a candidate, so a bare "0 dead" is never read as an all-clear.
- When the default method axis ran alone and found nothing, the note reports how many types in the scope the type axis would have judged and did not; the field `typesUnexamined` carries that count. A zero over an eligible population that still skipped some private methods as unanalyzed is qualified the same way.
- `testMethodsFiltered` and `testTypesFiltered` report candidates the default test filter removed, so a zero that is the filter's does not read like a zero that is the code's.
- Works on a recalled session.

`find_production_dead`

Finds production-dead symbols - public or internal types and methods that are used, but only by tests. This is the feature you built and tested but never wired into production. It is exactly what `find_dead_code` misses, because a symbol with test callers has a fan-in greater than zero.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`scope`	string	no	`solution`	`solution` or a namespace prefix.
`kinds`	string	no	`all`	`all` (methods + types), `method` or `type`.

Response. `scope`, `note`, `publicApiNote` (where it applies), `methodsScanned`, `candidatesWithoutAnyCaller`, `deadMethods`, `typesScanned`, `deadTypes`, `methodsIneligible` and `typesIneligible`. Each method carries `name`, `declaringType`, `namespace` and `testCallerCount`; each type carries `name`, `kind`, `accessibility`, `namespace` and `testReferenceCount`; either carries

`publicApi: true` when it belongs to a library's public API. Each axis is capped at 200.

Notes.

- The predicate is: direct callers exist, and the production impact is zero - the same production fan-in that `impact_of_change` reports. The test callers prove that the symbol is real and reachable rather than reflection-only noise; the production zero flags it as unwired.
- Methods are judged over their dispatch family: the method itself plus the same-named members of its declaring type's ancestry (base chain and interfaces, transitively). A call bound to `IFoo.Bar` records its edge on `IFoo.Bar`, so an interface member's production callers are seen and a concrete implementation is not a false positive, while a same-named method on an unrelated type neither lends evidence nor suppresses it. Where an interface has several implementers, the count can include tests that ran against a different one.
- Every judgement is made per declaration: the family starts from the candidate's own declaration and its own ancestry, and its production impact is the per-declaration closure `impact_of_change` reports, so a same-named type in another project or namespace neither lends callers to a candidate nor takes them away. A partial type split over several files is one row. A `partial class Program` in the global namespace - the declared half of a top-level-statements entry point - is an entry point, not a candidate. Nothing declared in a committed `*.Designer.cs` or `*.g.cs` file is reported.
- Dispatch and framework escapes are exempt: virtual, override and abstract members; well-known interface members such as `Dispose` or `Equals`; `On*` event handlers; `[RelayCommand]` and serialization-callback attributes; and whole XAML, test-scaffolding and framework-host types. Types use the same exemption taxonomy as `find_dead_code`, MEF parts included.
- **Library public API.** As in `find_dead_code`, a type of a library's public API carries `publicApi: true`, and so does a public method of such a type, or a protected or protected internal one where the type is neither sealed nor static. Nothing is exempted; `publicApiNote` (`LIBRARY PUBLIC API: ... methods and types carry publicApi:true ...`) says that a consumer outside the solution records no edge, so for them 'no production caller' covers this solution only.
- Uses from `.cshtml` views the Razor generator compiled are read here as in `find_usages`; when views were not compiled, a non-empty candidate list carries the `VIEWS NOT READ` sentence (see `find_usages`).
- Test-project symbols are the callers that matter here, never candidates themselves: a test method is not production-dead. A field initializer in a test class counts as test code, not as a production consumer.
- `candidatesWithoutAnyCaller` counts public or internal methods dropped because they have no caller anywhere in their dispatch family. Nothing proves them real, so they are not production-dead - but `find_dead_code` will not show them either, because its method axis is private-only. This number is the only sign that this class exists.
- The tool errs toward under-reporting: a symbol reached only via reflection, dependency injection or a source generator that static analysis cannot see is exempt. What a flag establishes with high confidence is the test usage; the production zero beside it is recorded, not proven, and the answer's own note says so. An empty result is not proof either.
- An empty answer is qualified only when it was structurally guaranteed: a scope that matched nothing says so instead of posing as clean, and a scope where no scanned unit was eligible reports "not judged here" with the ineligible counts.
- Works on a recalled session.

`find_god_objects`

Finds "god object" candidate classes - a class with too many members, a Single Responsibility Principle smell and a prime refactoring target. It is the queryable, scoped, member-count-ranked twin of the `quality-large-classes` quality finding, and it returns the per-class member breakdown so the split is actionable.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`minMembers`	integer	no	`25`	The member-count floor: only classes with at least this many members are listed. Values below 1 are raised to 1. Raise it, for example to 40, for only the worst offenders; lower it to explore.
`scope`	string	no	`solution`	`solution` or a namespace prefix.

Parameter	Type	Required	Default	Meaning
`includeTests`	boolean	no	`false`	Include test-project types.

Response. `scope`, `minMembers`, `classesScanned`, `matches` and `testMatchesFiltered`. Each match carries `name`, `namespace`, the total `members` and the breakdown `methods`, `properties` and `fields`. Ranked by member count descending, capped at 200.

Notes.

- Only classes are considered; records, structs and interfaces are excluded, as in the quality finding.
- The member count is the class's declared methods plus properties plus fields. A partial class is one row, its members summed over all its declarations, as in the `quality-large-classes` finding; `classesScanned` counts it once. Constructors, operators and source-generated properties are deliberately not counted.
- The floor is inclusive: a class with exactly `minMembers` members is listed.
- On a ranked answer, a non-zero `testMatchesFiltered` is a warning about the top of the list, not only about its length: the filter can remove the largest class of all while the remaining list still looks complete.
- Works on a recalled session.

`symbol_metrics`

Queries per-method complexity directly, or ranks the complexity hotspots. Both metrics are always returned: McCabe cyclomatic complexity (the number of independent paths) and SonarSource-style cognitive complexity (how hard the method is to understand: a nesting penalty, `else` and `catch` counted, boolean-operator runs collapsed).

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`symbol`	string	no	-	An exact, case-sensitive method name, bare (every same-named method) or qualified as `Type.Member` (that type's member only). Omit to rank all methods by complexity.
`minComplexity`	integer	no	`0`	When ranking: include only methods whose ranking complexity is at least this value. `0` means no filter.
`scope`	string	no	`solution`	When ranking: `solution` or a namespace prefix. Ignored when `symbol` is given.
`includeTests`	boolean	no	`false`	When ranking: include test-project methods. Ignored in the by-name lookup.
`rankByCognitive`	boolean	no	`false`	When ranking: sort and filter by cognitive complexity instead of cyclomatic complexity. Both numbers are always present per method; this only changes the axis. Ignored in the by-name lookup.

Response. `scope`, `mode` (`symbol` or `ranked`), `note`, `minComplexity`, `minComplexityBoundary`, `methodsScanned`, `methods`, `nearest` and `testMethodsFiltered`. Each method entry carries `name`, `declaringType`, `namespace`, `cyclomaticComplexity`, `cognitiveComplexity` and `parameterCount`, plus `project` where two projects compile the same type. Capped at 200 - the top entries when ranking.

Notes.

- The complexity floor is inclusive: a method whose ranking complexity is exactly `minComplexity` is listed, and a filtered answer states that in `minComplexityBoundary`. `solution_metrics` counts methods over its threshold strictly, so at the same number its count is smaller by exactly these boundary methods.

- The by-name lookup is never filtered by tests or by the floor, and it also resolves a user-defined operator, an accessor or a constructor by its metadata name (`op_Equality` , `get_Items` , `.ctor` - the last unions every constructor in the solution). The ranked hotspot view stays methods-only.
- A by-name lookup that matches no method carries a `nearest` suggestion, the closest declared name. If the exact name resolves to a declared type rather than a method, the note says so and points to `find_symbol` or `symbol_signature` . An empty ranking never steers: nothing above the complexity floor is a measurement, not a typo. The other way to an empty ranking is a `scope` that matches no namespace: `methodsScanned` is then `0` , and a `note` says where to look instead (the namespace of a type the prefix names, the namespaces that contain the prefix, or the closest namespace).
- `testMethodsFiltered` counts the test-project methods the filter removed; it is counted over the removed population, which `methodsScanned` does not include, so it can exceed that number. When the strongest filtered method would have landed inside the shown ranks, a note names it and the rank it would hold - a count alone cannot tell you whether the missing entries sat at the bottom or at the top.
- When the two complexity axes would deliver different sets of methods, a leading note names the axis that ordered this list and the methods the other axis would have shown. The note stays silent when they agree.
- Works on a recalled session. A session recorded before cognitive complexity existed reports `0` for it.

`solution_metrics`

Returns the solution-wide quality rollup in one call - the same numbers that the `--fail-on` quality gate evaluates.

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`failOn`	string	no	-	A gate expression in `--fail-on` syntax, for example `critical>0 OR debt>120min OR ce-max>50` . An invalid expression or an unknown metric is rejected with the list of valid tokens.
`dbPath`	string	no	-	Path to an AIContextBuilder database. When set, the complexity threshold and the producer switches come from that database's active quality profile. Omit to use the server's default configuration database.

Response. `metrics` , `scope` , `testMethodCount` , `testTypeCount` , `severities` , `debtMinutes` , `debtRating` , `complexityThreshold` , `complexityThresholdBoundary` , `qualityProfile` , `layerProfile` , `knownGateMetrics` , `gate` and `degradedProducers` .

- `metrics` carries `typeCount` and `methodCount` (logical types - partial fragments merged, multi-targeted projects deduplicated), `complexityMax` and `complexityAvg` , `methodsOverComplexityThreshold` , `ceMax` and `caMax` (efferent and afferent coupling maxima). A generic type parameter that shares its name with a solution type is not counted toward that type's afferent coupling - here, in `architecture_overview` 's most-coupled types, or in the fan-in tools.
- `severities` counts the quality findings by `critical` , `warning` , `info` and `ok` ; `debtMinutes` and `debtRating` are the technical-debt estimate, the rating a coarse letter from `A` (very low) to `E` (high).
- `complexityThresholdBoundary` states the boundary with the concrete numbers: `methodsOverComplexityThreshold` counts methods strictly above the threshold, while `symbol_metrics(minComplexity=...)` and the complex-untested finding use an inclusive floor.
- `knownGateMetrics` lists the valid metric tokens: `critical` , `warning` , `info` , `ok` , `debt` , `complexity-max` , `ce-max` , `ca-max` , `methods` and `types` .
- `gate` echoes the expression, `passed` and the `violatedClauses` when you passed `failOn` . It is advisory; the blocking gate remains the CLI.
- `degradedProducers` appears only when a producer failed during the run, in which case the severity counts and the debt figure are incomplete.
- `qualityProfile` and `layerProfile` disclose which profiles produced the numbers. On a server without a config DB and with no `dbPath` , a `configDbNote` adds that the built-in default quality profile judged - so a `qualityProfile` of `Default` is not mistaken for a database profile of that name.

Notes.

- The scope is production code only, matching the CLI gate: test projects are excluded from every number. The response says so itself with `scope: "production"`, and `testMethodCount` and `testTypeCount` carry the excluded test-side tallies in the same units - `methodCount` plus `testMethodCount` is the full solution's method-unit count.
- The rollup is deliberately not triage-filtered: insight suppressions apply to the reading surfaces (`list_insights` and `get_insight`), while this rollup and the CLI gate keep counting every finding. On a triaged solution these severity counts are therefore higher than what `list_insights` returns, and that gap is the suppression set.
- Works on a recalled session. The line-based debt and long-method producers degrade the same way as the CLI gate on a recalled snapshot; the metric rollup itself is stable.
- The tool writes nothing of its own; an older database passed as `dbPath` is upgraded on open, as the server upgrades its own at startup.

`detect_circular_dependencies`

Detects circular namespace dependencies - groups of namespaces that depend on each other in a cycle. This is a modularity smell that the C# compiler allows, unlike project-reference cycles, which it forbids. It complements layer-violation analysis with the orthogonal question of who is mutually entangled.

Availability: part of the default tool set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`scope`	string	no	`solution`	`solution` reports every cycle; a namespace prefix narrows the report to cycles that touch it. The analyzed graph stays the whole solution, so `namespacesScanned` does not change.
`includeTests`	boolean	no	`false`	Include test projects. Off by default, because a test-to-production reference is one-way and not an architecture cycle.

Response. `scope`, `namespacesScanned`, `cyclesFound` and `cycles`. Each cycle lists the participating `namespaces`, its `size`, the `projects` in which they are declared, `edgeCount` and the witness `edges`. Each witness carries `fromNamespace`, `toNamespace`, `fromType`, `toType` and `typeEdgeCount`, and the cycle carries `spansMultipleProjects`.

Notes.

- Each reported cycle is a strongly connected component of at least two namespaces, listing the participating namespaces plus witness type-to-type edges, one per directed namespace pair, that show why each link exists.
- Read the witnesses as examples, not as a work list: `edgeCount` counts directed namespace pairs, and each witness carries `typeEdgeCount` - how many distinct type-to-type references cross that one boundary. Where `typeEdgeCount` is 1, the example is the inventory; where it is higher, breaking the shown reference leaves the namespace edge standing and the next reference surfaces.
- A cycle inside one assembly is pure namespace organization and can be refactored freely. A cycle that spans assemblies means a namespace is reused across assembly boundaries - a different and often more telling situation, since project-reference cycles are compiler-forbidden.
- Namespace edges are derived from fully qualified facts, so a same-named type in two namespaces is never conflated, and external references are ignored. Only cycles inside the solution are reported.
- A clean, well-layered codebase returns zero cycles. That empty result is itself a useful signal.
- Cycles are capped at 100; assembly-spanning cycles come first, then the largest tangles. Each cycle's namespace list stays complete; only the witness edges are capped.
- Works on a recalled session.

8.3 Patterns and documentation drift

`find_structural_twins`

Finds symbols with the same structure as a type or method - the siblings a text search cannot see. Use it before editing one member of a family to find the others you must keep consistent.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`symbol`	string	yes	-	The exact, case-sensitive type or method name to find twins of.
`minSimilarity`	number	no	`0.8`	Minimum structural similarity, from 0 to 1. Higher is stricter; about 0.9 is near-identical, 0.6 widens the net, and 0.7 floods on a large codebase. Values outside 0 to 1 are clamped, and the answer says so.

Response. `symbol`, `resolvedKind`, `minSimilarity`, `twins` and `note`. Each twin carries its `name`, a 0 to 1 `similarity` score, the `sharedDimensions` and `divergingDimensions`, its `namespace` and - on the method axis - the `declaringType` it is declared on. Capped at 100.

Notes.

- The comparison uses the declared shape, not what the bodies do: two methods with the same signature score high even when their implementations are unrelated.
- Both axes need a structural anchor. A type needs a base class or a non-ubiquitous interface; a method needs at least one modifier or a parameter of a non-ubiquitous type - `string`, the primitives, `object`, collection heads and `CancellationToken` are too common to mark a family. Without an anchor the signature collapses onto ubiquitous dimensions, where every base-less record or every parameterless `private void X()` would score 1.0. Such a seed returns no twins plus a note; that note means "structurally generic", not "no twins exist".
- Candidates above the threshold that share only the ubiquitous skeleton and carry no matching distinctive signal are suppressed as noise. The note reports how many out of how many - `totalFound` is the count after suppression, so do not subtract the two. Lowering `minSimilarity` will not turn suppressed candidates into twins.
- The seed is resolved by bare name, and there is no qualified `Type.Member` form. When the name has several declarations (partial fragments count once), the answer describes the first in enumeration order and the note says so, naming the other owners where they differ. Overloads on one type share an owner, so the note then states the count alone; use `symbol_signature` to see each declaration with its file.
- Test-project symbols are excluded from the candidates: structural twins are a production concern.
- If `minSimilarity` was outside 0 to 1, the note names the value you sent and the value that was used, and states that the whole answer describes the clamped threshold. A decimal comma is the usual cause on a non-English desktop.
- Works on a recalled session.

`check_pattern_drift`

Given a set of symbols that should follow the same pattern - for example all your panel view models - computes the consensus shape and pinpoints which member deviates and in what dimension.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.

Parameter	Type	Required	Default	Meaning
`symbols`	array of strings	yes	-	The type or method names that should share a pattern. Two are the minimum, three or more give a real consensus. Unknown names are reported, never silently dropped. A name several declarations share is measured as the first of them; a qualified name (`Type.Member` , `Ns.Type.Member` , `Ns.Type`) picks one type, but not one overload.

Response. `consensusKind` , an optional `note` , `memberCount` , `conformingCount` , `consensusScalars` , `consensusSets` , `deviations` and `noConsensusScalars` .

- `consensusKind` is the majority kind (`type` or `method`), or `not_found` when none of the names resolved.
- `note` stands ahead of `memberCount` , the number it qualifies: it names repeated names (counted once) and names several declarations share (measured as the first only).
- `consensusScalars` and `consensusSets` are the published consensus shape: scalar dimensions with their shared value, set dimensions with their shared elements.
- `deviations` lists the members that differ from the consensus; each entry carries the `member` , its `resolvedKind` and the `deviations` as human-readable dimension lines. A member of the minority kind is flagged as a kind deviation, and an unresolved name is reported with `not_found` .
- `noConsensusScalars` lists scalar dimensions that were deliberately left out because no single value reached a majority, each with its observed distribution.

Notes.

- Consensus means majority: a dimension is published only when one value (scalar) or one element (set) is held by more than half of the members. A scalar on which the members merely disagree - three types in three different size buckets, say - is not published on a plurality or a tie. It is listed in `noConsensusScalars` instead, because reporting members against a shape none of them holds would manufacture the deviations this tool is asked to find.
- Read `noConsensusScalars` before `conformingCount` : "all conform" plus a dropped dimension means "they agree on the rest and share nothing there", not "they follow one pattern".
- A dimension that no member carries at all is passed over in silence rather than reported as a no-majority finding.
- Works on a recalled session.

`check_doc_drift`

Finds documentation drift - XML documentation comments that no longer match a method's actual signature.

Availability: not part of the default tool set; use the `Full Select` profile or the `AICB_MCP_TOOLS` override.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id or a solution path.
`scope`	string	no	`solution`	`solution` or a namespace prefix.

Response. `scope` , `methodsScanned` , `methodsWithDocTags` and `findings` . Each finding carries the method's `name` , `declaringType` , `namespace` , its check `kind` and a human-readable `detail` . Capped at 200.

Notes.

- Two reliable, low-false-positive checks are performed. `param-not-found` : a `<param name="x">` whose name is not an actual parameter - a renamed or removed parameter, or a typo in the documentation. `return-on-void` : a `<returns>` on a method that returns `void` .
- A `<returns>` on `Task` , `Task<T>` , `ValueTask` or any other value-returning method is correctly not flagged: documenting the returned task is a standard convention, not drift.
- `methodsWithDocTags` counts the methods that carry a `<param>` or `<returns>` tag at all - the drift-checkable ones, not the summary-documented ones. Methods without such tags are skipped; there is nothing to check.

- Live only: this tool reads documentation facts that are not stored with a session, so it needs a live analysis. A recalled session is rejected with a pointer to refresh it.

9 Tool reference: markup, export, review and insights

This chapter documents the tools for XAML/AXAML markup, for packing and exporting context, for reviewing changes, and for reading quality insights. All of them are called through an MCP client. Two of them can write: `export_markdown` writes a file when you pass `outputPath`, and `diff_review` writes to the triage log when you pass `recordRegressions: true`. Everything else only reads.

Several tools accept a `dbPath` for the configuration database. When you omit it, the server's default configuration database applies - the same database the desktop application uses - so the quality profile and layer profile you configured there take effect. For `diff_review` with `recordRegressions: true`, that default also determines where the write goes.

Most tools take a `sessionId`. A session is created by `analyze_solution`, or on first use when you pass the absolute path of a `.sln`, `.slnx` or `.slnf` file; see "Sessions and staleness".

The default MCP profile exposes a curated tool set. Where a tool below is not part of it, its availability line says so and names the two ways to expose it: start the server with the `Full Select` profile (`aicb mcp --mcp-profile mcp-profile/full`), or set the `AICB_MCP_TOOLS` environment variable. Where a tool can be used as a sub-query of `batch`, the availability line says that too.

9.1 Markup: XAML/AXAML bindings and resource keys

Markup connects views to view-models and to resources by string. Resource keys have no C# symbol edge, while a resolved view-model's root `{Binding}` member is represented in the symbol graph as a low-confidence `(xaml)` edge; the graph does not retain the markup site's binding form. `find_resource_usages` and `find_binding_usages` scan current files on disk and therefore do not need `refresh_session`. `find_unresolved_bindings` instead reads the live analysis result: refresh after markup or view-model edits, and expect no findings from a recalled snapshot.

`find_unresolved_bindings`

Purpose. List silently broken bindings: a `{Binding X}` whose root member does not exist on its confidently resolved DataContext view-model - a typo, a removed or renamed member, or the wrong DataContext. The compiler does not catch this class of defect.

When to use it. After renaming or removing a view-model member, after moving a view to another view-model, and as a lint pass before a release.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution` (or the absolute path to a solution file).

Coverage.

- WPF/WinUI `.xaml`: the view-model is resolved through `d:DesignInstance` or through a unique `FooView` -> `FooViewModel` naming convention.
- Avalonia `.axaml`: `{Binding}` and `{CompiledBinding}`; the view-model is resolved only through `x:DataType` at scope level, so no naming convention applies.
- A nested type written `prefix:Outer+Inner` in a `DataType` / `x:DataType`, written as `{x:Type ...}` or in the bare form, scopes its bindings to the nested type, not to its outer type.

What is deliberately skipped. A binding whose DataContext scope cannot be typed with certainty is never flagged: `ControlTemplate` / `Style`, a runtime DataContext, keyed resources, and a template without a `DataType`. WPF alone recovers an item template's type from the host's pinned `ItemsSource`; on Avalonia, a template without `x:DataType` - `TreeDataTemplate` included - is always skipped. Only view-models resolved with certainty and with a verifiable member surface are checked; source-generated members (`[ObservableProperty]`, `[RelayCommand]`) and inherited members count as resolved. The trade-off is a recall miss, never a false report.

Response.

Field	Meaning
`count`	Number of unresolved bindings found.
`unresolvedBindings`	One entry per finding: `view`, `viewModel`, `member` (the absent root member), `line` (1-based line of the binding attribute; `0` means unknown).
`markupFilesScanned`	Markup files discovered by a live disk walk of the session's project directories.
`bindingSitesInMarkup`	Binding sites found by the same live walk - the markup population.
`sitesJudged`, `sitesSkipped`	The denominator of `count`: the binding sites whose root member was checked on a resolved type, and the sites the tool skipped. `null` on a session without a live verdict (a recalled snapshot); `0` there only when the live walk found no binding site at all.
`sitesSkippedBy`	The skipped sites, counted per reason. Omitted when nothing was skipped.
`judgementNote`	Present only when the judged population differs from `bindingSitesInMarkup`, naming each part of the difference with its number: element-syntax `<Binding Path="X"/>` sites (counted as judged or skipped, but not in `bindingSitesInMarkup`, which lists the `{Binding}` form only), markup edited since the analysis (call `refresh_session`), and sites that could not be attributed, with how many of them are element-syntax sites. On a recalled session it says why both counts are `null`.
`note`	Present when the zero needs qualification: a session without a live binding verdict (a recalled snapshot), an incomplete analysis run, or a solution with no markup at all.

Note: `bindingSitesInMarkup` is the live scan total, not the denominator of `count` - it includes the scopes this tool conservatively skips. `sitesJudged` and `sitesSkipped` are the decisions of the same analysis pass that produces the findings, so a zero has a denominator. Read a hit as high-confidence evidence that you verify at its `file:line`, not as proof.

Note: identical findings from multi-targeted analysis instances are deduplicated, so a site is reported once, not once per target framework.

Note: the verdict is computed live during analysis and is not persisted. A session recalled from a saved snapshot therefore reports none, with `sitesJudged` and `sitesSkipped` at `null`, and the `note` says so. Call `refresh_session` after markup or view-model edits and run the tool again.

Note: the same findings also appear in `list_insights` as the `design-unresolved-xaml-binding` insight, so a review of that insight sees them without calling this tool.

Availability: in the default tool set; usable as a sub-query of `batch`.

`find_resource_usages`

Purpose. Resource-key fan-in across the solution's XAML/AXAML markup and its C# sources - the question the C# symbol graph cannot answer, because resources are wired by string key.

When to use it. Before deleting or renaming a resource key. Deleting a still-referenced key compiles cleanly and throws only at runtime, so a text search is the wrong instrument: this tool also finds the references that live in C# string literals.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution`.
`key`	string	no	`null`	The exact, case-sensitive resource key, e.g. `TextPrimaryBrush`. Omit it for the audit view.
`scope`	string	no	`"solution"`	`solution`, or a view-path substring (e.g. `AIContextBuilder.UI/`) to narrow which files are reported.

With a key.

Field	Meaning
`definitions`	Where the key is defined (`x:Key`), each with `view` and `line` .
`definitionCount`	Number of definitions in scope.
`references`	Every reference site with `view` , `line` and `kind` .
`referenceCount` , `staticReferenceCount` , `dynamicReferenceCount` , `codeReferenceCount`	Reference totals, split by channel.
`themeReferenceCount` , `callerMemberNameReferenceCount`	The totals of the two further channels below; present only when not zero.
`sameFileCollisions`	Duplicate definitions of the same key in one file (`key` , `view` , `lines`).
`referencesTruncated`	`true` when the reference list was capped at 200.
`scannedFileCount` , `scannedCodeFileCount`	Markup files and C# files read.
`note`	Present when there is something to qualify (see below).

The `kind` of a reference tells you which channel it came through:

- `static` - `{StaticResource Key}` , resolved once at load time.
- `dynamic` - `{DynamicResource Key}` , re-resolved on every lookup, so it survives a theme swap. A brush that is swapped with the theme must be referenced dynamically.
- `theme` - WinUI's `{ThemeResource Key}` .
- `code` - a C# string literal whose text equals the key, the `TryFindResource` channel, reported with its file and line.
- `callerMemberName` - a C# member whose name is the key and that reads it through a `[CallerMemberName]` accessor indexing a resource dictionary or calling `FindResource` with that parameter, reported at the call site.

A markup extension the solution itself derives - directly or through its own types - from `StaticResourceExtension` or `DynamicResourceExtension` (for example a `{ui:ThemeResource Key}`) counts as its base's kind, `static` or `dynamic` .

Only same-file duplicates are reported as collisions: in a compiled resource dictionary a later duplicate silently shadows an earlier one, while a same-named key in a *different* file is usually a legitimate theme pair.

The `note` distinguishes four states: the key is not found anywhere in the markup (misspelled - keys are case-sensitive - or it lives in a theme or library assembly); the key is defined but has no references; the key is referenced but has no definition in scope (it may be defined in a theme or library assembly); or the scan read no markup at all.

Without a key (audit view).

Field	Meaning
`totalDefinedKeys`	Number of distinct keys defined in scope.
`neverReferencedKeys`	Every key with zero references, grouped with its definition sites (`key` , `definedIn`): the deletion candidates.
`neverReferencedTotal` , `neverReferencedTruncated`	True total and cap flag (list capped at 200).
`sameFileCollisions`	All same-file duplicate definitions in scope.
`scannedFileCount` , `scannedCodeFileCount`	Markup files and C# files read.
`note`	The fan-in scope note (see below), or the no-markup note.

Scope. In the audit view the scope narrows which definitions are audited; references still count solution-wide, so a key referenced outside the scope is not a false orphan. With a key, the scope narrows both the definitions and the references that are reported, and the counts follow.

Note: the tool reads the current `.xaml` / `.axaml` / `.cs` files on disk (`bin` , `obj` and `.vs` are excluded). It is never stale after an edit, needs no `refresh_session` , and works on recalled sessions.

Note: only string keys are in scope. A structural `x:Key="{x:Type ...}"` is not seen; the long form `{StaticResource ResourceKey=...}` and the element `<StaticResource ResourceKey="..." />` are read. XML-commented markup is ignored.

Note: fan-in comes from the solution's markup and its C#. In markup: `{StaticResource}` , `{DynamicResource}` , WinUI's `{ThemeResource}` and the derived extensions above. In C#: string literals equal to the key, and members read through a `[CallerMemberName]` accessor (matched by name, not by type). A matching literal is not by itself proof of a resource lookup: it may be a test assertion or a documentation `cref` naming a same-named converter type. Still invisible are a key assembled at runtime (interpolated or concatenated), a source file that is not `.cs` , a resource-extension subclass that ships in a referenced library, and a third-party library template that resolves the key from its own assembly. Zero references means "verify before deleting", not proof.

Availability: in the default tool set; usable as a sub-query of `batch` .

`find_binding_usages`

Purpose. Binding-path fan-in across the solution's markup. The symbol graph records a `{Binding}` root member against a resolved view-model as a low-confidence edge tagged `(xaml)` ; this tool adds the site list, the resolution form, the deeper-path split and the forms the fan-in layer skips.

When to use it. Before renaming or deleting a bound member, to see every markup site that consumes it - and to see which bindings hang on an ancestor lookup instead of on the DataContext.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution` .
`path`	string	no	`null`	The exact, case-sensitive bound member path, e.g. `GestureLabel` or `Foo.Bar` . Omit it for the inventory view.
`scope`	string	no	`"solution"`	`solution` , or a view-path substring (e.g. `AIContextBuilder.UI/Views/Dialogs`) to narrow which markup files are reported.
`form`	string	no	`null`	Form filter: `dataContext` , `elementName` , `relativeSource` or `source` . Omit it for all forms. Applies to both views.

With a path.

Field	Meaning
`sites`	Every binding site: `view` , `line` , `path` , `form` .
`siteCount`	True number of sites (the list is capped at 200).
`byForm`	Site counts per form: `dataContext` , `elementName` , `relativeSource` , `source` .
`deeperSegmentSites` , `deeperSegmentSiteCount`	Sites that name the path only as a deeper segment of another member's path (see below).
`pathlessBindingCount`	Bindings that name no member at all - solution-wide, not narrowed by `scope` or `form` .
`sitesTruncated`	`true` when the site list was capped at 200.

Field	Meaning
`note`	The matching result, the deeper-segment disclosure and the fan-in scope note.

The `form` of a site tells you what it resolves against:

- `dataContext` - the view-model; the only form that is a view-model member edge.
- `elementName` - another named element in the same scope.
- `relativeSource` - an ancestor, the templated parent, or self.
- `source` - an explicit object (a static, a resource).

Matching rule. A path matches as the whole path or as its root segment. `Foo` finds `{Binding Foo}` and `{Binding Foo.Bar}` - both consume `Foo`. The `Bar` in `{Binding Foo.Bar}` does not match: it is a member of `Foo`'s type. Such sites are reported separately under `deeperSegmentSites` with their own count, because a rename of the same-named member on the nested type goes through exactly them. When there are deeper-segment sites, the `note` names up to two of them and points at the list.

Element hop rule. On an `elementName` or `relativeSource` redirect, a leading `DataContext.` is the hop to the element's view-model, not a member: `{Binding DataContext.Cmd, RelativeSource={RelativeSource AncestorType=UserControl}}` - the item-template idiom for a command on the page view-model - is a site of `Cmd`. On a plain binding or a `Source=` binding the segment stays the root, because there the source is not an element by construction. A hop in the middle of a path, such as `PlacementTarget.DataContext.Cmd` in a ContextMenu, is not followed either; such a site roots at `PlacementTarget`.

Without a path (inventory view).

Field	Meaning
`members`	Every distinct bound root member with `member`, `siteCount` and `byForm`, most-bound first. `{Binding Foo}`, `{Binding Foo.Bar}` and the hop form are one entry.
`distinctMemberCount`, `totalSiteCount`	Distinct members and total sites in scope.
`pathlessBindingCount`	Bindings that name no member - solution-wide.
`membersTruncated`	`true` when the member list was capped at 200.
`note`	The fan-in scope note, or a note about an unknown form filter.

`pathlessBindingCount` covers an empty `{Binding}`, a parameters-only `{Binding Mode=OneWay}` and an attached-property path, so the listed sites and the markup's binding count differ visibly rather than silently.

Note: an unrecognized `form` value matches nothing rather than everything - widening a typo to "all forms" would answer a question nobody asked. The `note` names the typo and the valid values, so the zero is not read as a finding.

Note: this is markup-only fan-in. It sees the `{Binding}` markup-extension form in the solution's own `.xaml` / `.axaml` files. A binding built in code-behind (`SetBinding`, `BindingOperations`), the element syntax `<Binding Path="..."/>`, a WinUI `{x:Bind}`, a `{TemplateBinding}` and any path assembled at runtime are invisible here - zero sites means "verify before renaming or deleting", not proof. This scan also does not resolve the path against a DataContext; `find_unresolved_bindings` is the tool that checks whether a bound member actually exists on its view-model.

The tool reads the current files on disk, so it is never stale after an edit, needs no `refresh_session`, and works on recalled sessions.

Availability: in the default tool set; usable as a sub-query of `batch`.

9.2 Packing and exporting context

Four tools with overlapping purpose. The order below is the decision aid: read one symbol with `get_context`; read one symbol with a chosen environment with `explain_symbol`; bundle a task from a goal, read-only, with `pack_for_task`; bundle the same task edit-ready with `prepare_task`; render the whole solution with `export_markdown`.

`pack_for_task`

Purpose. Goal-driven context packing: given a natural-language task, seed on every symbol the goal names (type or method), expand the neighborhood, and return a token-budgeted AI-Builder-Markdown bundle with goal-aware trimming that keeps the most task-relevant methods when the budget is tight.

When to use it. You have a task description and want the relevant code in one answer, without the extra material `prepare_task` adds.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution`.
`goal`	string	yes	-	The task description. Name the relevant types and methods in it, e.g. `refactor OrderService.Cancel and Validate`. A member written as `Type.Member` seeds that type's member only (a member the type inherits: the nearest base class or interface that declares it); a member qualified by something that is not a type (a variable, a namespace) is read as if it were written bare.
`budget`	integer	no	`null`	Token budget (floor 8000). Exact type names and exact method names with one owner are pinned; broad subword or ambiguous same-name families may be capped by closeness and in-slice fan-in. A leading comment discloses how many types were dropped. Omitted, the bundle renders against a ceiling: the active MCP profile's token budget, else about 10000 tokens.
`includeQualityMetrics`	boolean	no	`false`	When true, also emit the `<QUALITY_HOTSPOTS>` section plus `complexity=""` and `ce=""` attributes for the packed symbols. A type's afferent coupling (`ca=""`) counts the referencers in the whole solution, not only those in the bundle.
`lean`	boolean	no	`true`	Drop the repeated format-rules preamble and omit enabled-but-empty graph sections; keeps the legends and all non-empty sections. Set `false` for the full AI-Builder-Markdown document.
`facet`	string	no	`null`	The task axis of this call: `General` (default), `Exploration`, `Refactoring`, `Debugging`, `Review`, `Testing`, `Documentation`, `Architecture`, `Performance`. The former name `slot` is still accepted.

Note: `pack_for_task` always renders the lean bundle - its render deliberately stays template-free. The facet only appends a short work-style trailer with suggested next tools. For a template-shaped bundle, use `prepare_task`.

Response. The Markdown bundle. If the goal names no known symbol, the answer is an HTML comment that says so (`pack_for_task: goal mentions no known symbol - try naming a type or method`) rather than an empty document.

Note: on a multi-targeted solution, the bundle is deduplicated to one logical project per name, using the newest target framework's instance - the same view `find_symbol` uses.

Note: when several types share a name the goal names, every one of them renders. A same-named type that the ranking left out is not added.

Note: a member qualified by an interface (`IOrderService.Cancel`) also seeds that member on every production type that implements the interface - where an implementation inherits the member, on the base class that declares it. A class qualifier does not hand its interfaces on to other implementations. The goal's own names rank first; the seeds are capped at 24 declarations.

Note: the bundle renders as `get_context` does: a type the budget reduces to structure lists the signatures of the methods it does not render as method blocks; a goal naming an interface pulls its implementing types, for a generic interface also those that close it through a base class or another interface; and a partial type renders one block per project it is compiled into.

Availability: in the default tool set; usable as a sub-query of `batch` .

`prepare_task`

Purpose. A task-ready context bundle: seed on the symbols the goal names, expand the neighborhood, and additionally add the test methods that cover those symbols plus one or two naming or file-convention siblings - the test, the factory, the validator you would otherwise miss. Prefer this over `pack_for_task` when you are about to edit, not just read.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution` .
`goal`	string	yes	-	The task description; name the relevant types and methods in it.
`facet`	string	no	`null`	The task axis (same values as `pack_for_task`). On a server with a config DB - a plain `aicb mcp` resolves the standard one; `--db-path` names another - the facet's template shapes the render (`render: template` , the default); only a server without a config DB renders `lean` . A work-style trailer is always appended.
`budget`	integer	no	`null`	Token budget (floor 8000). Applies to both render paths. The goal-named seeds are always kept; the bundled covering tests and siblings are budget-bound and are dropped under a tight budget, with a leading comment naming the drops. Precedence: this parameter, then the MCP profile, then the template, then the global default. Omitted, the bundle renders against a ceiling: the profile's token budget, else about 10000 tokens.
`includeTests`	boolean	no	`true`	Include the test methods that cover the seeded symbols.
`includeSiblings`	boolean	no	`true`	Include one or two naming or file-convention siblings of the seeded types.
`includeQualityMetrics`	boolean	no	`false`	Emit the `<QUALITY_HOTSPOTS>` section and `complexity=""` / `ce=""` attributes. Applies to the lean render path only; on a default server, which renders the template path, it has no effect.
`lean`	boolean	no	`true`	Lean slice on the lean render path. Ignored on the profile-aware slot-template render path, which the active profile controls; on a default server, which renders the template path, it has no effect.

Response. The Markdown bundle, led by a one-line manifest so you can see what was covered - and what was not:

```
<!-- prepare_task | goal: "..." | seeds: ... | tests: ... | siblings: ... | render: template -->
```

`seeds` names the seeded symbols; a member bound to one type reads `Type.Member` . When the seed cap cut matching declarations, the segment ends in `(+N more matching declarations, capped at 24 - name one in the goal to seed it)` . The goal-naming rules of `pack_for_task` apply here too, including the seeds an interface-qualified member hands on to its implementations, and the covering tests follow the seed: a `Type.Member` seed bundles the tests of that type's member only.

Each bundled test carries its evidence tier, the `matchReason` that `find_tests_for` prints (`invokes`, `invokes-via`, `accesses`, `accesses-via`, `constructs`, `constructs-via`, `dispatch`, `dispatch-via` or `name`; see its tier table), and the list is ordered strongest tier first, then by name. When no row in the list is strong evidence, the manifest says so (`none invokes or constructs the symbol`). At most 40 covering tests are bundled per bundle; a cap is disclosed as `(+N more, capped)`, and when some of the tests found reach a seed only through an interface or abstract member it implements, the note adds how many of all found do: `(+N more, capped; X of Y reach a seed only through an interface or abstract member it implements)`. An axis that found nothing reads `(none)`. The `render` segment reads `template` on a server with a config DB, the default, and `lean` without one.

When the template the render runs through sets `Expansion around the MCP start point`, the manifest adds `| seed expansion: <Name> (<id>)`; a strategy id that does not resolve adds `| seed expansion: '<id>' not found, the default ran`, and one that cannot be read `| seed expansion: '<id>' could not be read, the default ran`. Without the setting, both render paths grow the neighborhood around the seeds the same way. A running server reads a template change only after a restart.

Note: on a multi-targeted solution, the bundle is deduplicated to one logical project per name, using the newest target framework's instance.

Note: when several types share a name the goal names, every one of them renders. A same-named type that the ranking left out is not added.

Note: on both render paths, a type the budget reduces to structure lists the signatures of every method it declares that does not already render as a method block (up to 80 lines, then `(+N more)`), and where quality metrics are rendered its `members` counts what the type declares in its project. On the template path, a type present in several projects (a linked file, a namesake, each target framework of a multi-targeted project) lists and counts only its own project's declaration in each block. On the lean path, the bundle also follows `get_context` for interfaces (implementing types pulled in, through a base class or another interface too) and for partial types (one block per project).

Availability: in the default tool set; not usable as a sub-query of `batch` (it is a heavyweight renderer).

`export_markdown`

Purpose. Render the cached analysis of a session as AI-Builder-Markdown. No re-analysis - the answer is instant.

Note: this renders the whole solution. On a large codebase the default render is a multi-MB document that can exceed the response or token limit. For a bounded view prefer `architecture_overview` (whole-solution orientation, structural only, no per-type source) or `get_context` / `explain_symbol` (one symbol's world).

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution`.
`facet`	string	no	`null`	The task axis of this call (same nine values as `pack_for_task`). The facet picks the active MCP profile's matching template and appends a short work-style trailer. Unknown or empty falls back to `General`; a facet whose slot has no template assigned falls back to the active context template. The former name `slot` is still accepted.
`outputPath`	string	no	`null`	Absolute path to write the Markdown to (UTF-8, no BOM). When given, the tool writes the file and returns a short confirmation - `Wrote N chars of AI-Builder Markdown to '<path>'. ...` - instead of the full Markdown, so a large render does not flood the response. A missing target directory is created.
`format`	string	no	`null`	The inner notation of the rendered Markdown: `tag` (the established AI-Builder tag format) or `yaml` (idiomatic YAML - a lossless re-notation, not a different selection of content). An explicit value wins; when omitted, the active MCP profile's `OutputFormat` applies on a profile-aware server, otherwise `yaml`. The `.md` file extension is unchanged.

Note: a `format` value that is neither `tag` nor `yaml` counts as omitted - the profile's format applies - and the answer starts with a note naming the ignored value (`Note: format '<value>' is not a known value ('tag' or 'yaml') and was ignored ...`).

Note: on a profile-aware server (a plain `aicb mcp` , which resolves the standard config DB, or one started with `--db-path`), the active MCP profile drives the render - the chosen facet's template (detail presets, graphs, line numbers, quality metrics) is used. Without a profile, the full default render is used.

Note: the work-style trailer is appended to the response only; it is never written into the `outputPath` file.

Response. The Markdown, or the short confirmation when `outputPath` is set.

Availability: in the default tool set; not usable as a sub-query of `batch` (it writes when `outputPath` is set).

9.3 Diff, review and change gates

`semantic_diff`

Purpose. Diff two analyzed sessions structurally: added and removed types and, for types present in both, added and removed methods plus method-signature changes (return type, parameters, accessibility, `static`).

When to use it. Analyze the same solution twice - for example before and after a refactor - and compare the two states. Both arguments are session ids; analyze the two states first.

Parameter	Type	Required	Default	Meaning
`sessionA`	string	yes	-	The "left" (baseline) session.
`sessionB`	string	yes	-	The "right" (changed) session.

Response.

Field	Meaning
`addedTypes` , `removedTypes` , `changedTypes`	Each a capped envelope with `items` , `count` , `totalFound` and `truncated` ; capped at 100 entries per list.
`changedTypes[].addedMethods` , `.removedMethods`	Method names added to or removed from a type that exists in both states.
`changedTypes[].changedMethodSignatures`	One entry per method whose signature changed, with `methodName` , `leftSignature` , `rightSignature` and the flags `returnTypeChanged` , `parametersChanged` , `accessibilityChanged` , `staticChanged` .
`totalChangeCount`	Sum of the three lists.
`isIdentical`	`true` when all three lists are empty.

Note: types are matched by fully qualified identity, so two same-named types in different namespaces are distinguished, not conflated. A removed type whose simple name is ambiguous solution-wide is rendered by its full name. The diff is syntactic - it compares types, methods and signatures, not method bodies.

Availability: not in the default tool set - the default profile withholds the two-session comparison family from its tool menus. Expose it with the `Full Select` profile (`aicb mcp --mcp-profile mcp-profile/full`) or `AICB_MCP_TOOLS` . Not usable as a sub-query of `batch` (it needs two sessions).

`diff_review`

Purpose. Review a change end to end in one call:

1. the structural diff of two analyzed sessions (before → after);

2. the blast radius of the added and changed types - for each, its fan-in usages, transitive change impact and risk, implementations, and covering tests;
3. the introduced findings - the code-quality, design-smell and layer-violation findings that exist in "after" but not in "before" (a delta, so pre-existing debt is never blamed), with a `pass` / `warn` / `block` verdict.

It is the natural "review my change" bundle: `semantic_diff` plus `review_context` plus the introduced-findings delta.

Parameter	Type	Required	Default	Meaning
`sessionBefore`	string	yes	-	The "before" (baseline) session id. Must be live.
`sessionAfter`	string	yes	-	The "after" (changed) session id. Must be live.
`policy`	string	no	`null` (= `block_on_critical`)	`block_on_critical`, `block_on_warning` or `advisory`. An unknown policy is rejected, not silently treated as `advisory`.
`dbPath`	string	no	`null`	The active QualityProfile (producer thresholds and toggles) and the configured default layer profile of that database. Omitted: the server's default configuration database, else the default profile.
`recordRegressions`	boolean	no	`false`	When true, each introduced finding is also written to the database's triage log as a `regression` event on its `(producer, type, member)` scope. This is the one thing this tool does that is not read-only, which is why it is opt-in and off by default.

Note: both sessions must be live. A session recalled from memory is rejected, because the findings delta needs equal-fidelity analysis on both sides - a recalled session lacks the live-only smell and metric facts, so a live-versus-recalled delta would falsely report all of the after state's live-only findings as introduced.

The policy verdict.

Policy	Verdict
any	`pass` when no finding was introduced.
`advisory`	`warn` whenever a finding was introduced; never blocks.
`block_on_warning`	`block` when the highest introduced severity is warning or critical; otherwise `warn`.
`block_on_critical` (default)	`block` when the highest introduced severity is critical; otherwise `warn`.

The findings delta honors the after session's explicit layer profile and its resolved test-detection profile; both delta sides are always generated under the same axes, so a configuration difference is never blamed as a code difference. Passing `dbPath` additionally unlocks that database's active QualityProfile and its configured default layer profile (per solution, then application-wide); an explicit layer profile passed to `analyze_solution` still wins.

`recordRegressions` **in detail.** When true, each introduced finding is recorded as a `regression` event in the database's triage log, so "this came back" survives a restart. At most one open regression is recorded per scope, so re-running the same review adds nothing. When no database is configured, or the database has never seen this solution, nothing is written - and the call still succeeds, with the reason in `regressionLog.skippedReason`; the solution row is never created here. You are asserting the direction: findings are recorded as introduced by `sessionBefore` → `sessionAfter`.

Response.

Field	Meaning
`diff`	The structural diff, in the same shape as `semantic_diff` (each list capped at 100).

Field	Meaning
`changedSymbolReviews`	The blast-radius reviews, capped at 50 changed symbols; the envelope carries the true total and `truncated`. Each review has the fields of `review_context`.
`introducedFindings`	The gate verdict: `result`, `reason`, `policy` and `introduced`.
`regressionLog`	Present only when `recordRegressions: true` was passed: `recorded`, `alreadyOpen`, `skippedReason`.

Availability: not in the default tool set; expose it with the `Full Select` profile or `AICB_MCP_TOOLS`. Not usable as a sub-query of `batch` (it needs two sessions).

`review_context`

Purpose. Assemble review context for a set of changed symbols: for each symbol, its fan-in usages, the transitive change impact plus a risk level, the types that implement it (if it is an interface), and the test methods that likely cover it.

When to use it. Before reviewing or refactoring: pass the type and method names you touched - for example the symbols in a diff - and see blast radius and test coverage at a glance.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution`.
`symbols`	array of strings	yes	-	The changed type or method names to assemble review context for. At least one is required.

Response. One review per symbol, each with:

Field	Meaning
`symbol`	The name you passed.
`note`	Present when the name resolved to nothing (the entry then carries no risk), or when the risk was graded on the direct fan-in or on the implementing types alone.
`resolvedKind`	What the name resolved to: `type`, `method`, `property`, `field`, `event`, `enum_member`, `not_found`.
`directUsages`	Number of direct users or callers - the `directCount` of `impact_of_change`, one per declaration.
`transitiveImpact`	Number of dependents through chains - the `transitiveCount` of `impact_of_change`, one per declaration.
`risk`	`none`, `low`, `medium` or `high`, graded as `impact_of_change` grades it - test callers count toward the usage numbers but not toward the risk.
`implementations`	The types implementing the symbol, when it is an interface - for a generic interface also those that get it through a base class or another interface. Test doubles are listed too; `find_implementations` filters them by default.
`invokesTotal`	How many of the covering tests are strong matches.
`dispatchTotal`	How many reach the symbol through an interface or abstract member. Omitted at 0.
`coveringTests`	The covering test methods, each with its evidence tier.
`coveringTestsTotal`, `coveringTestsTruncated`	The full number of covering tests, and whether the list was cut.

Note: `coveringTests` is the `find_tests_for` list under the same contract - strongest evidence first - and `invokesTotal` names how many of them are strong. Ranking by position is therefore meaningful, and you can see where the evidence stops.

Note: the list is capped at 50 per symbol and keeps the 50 strongest rows: name guesses go first, strong rows too once there are more than 50. `invokesTotal` and `dispatchTotal` count the whole list and can exceed the rows shown; `coveringTestsTotal` minus both is the number of name guesses. `find_tests_for` stops at the same 50, so narrow the symbol rather than re-query.

Note: when the risk stands only on the implementing types - no production code calls the interface or abstract member - the row's `note` says so with the same sentence `impact_of_change` puts in `riskBasis.note`. For a bare name several types share, the counts are their union and no note says so; `find_usages` names the namesakes. For a qualified method, `directUsages` and `transitiveImpact` include callers credited through the interface or abstract members it implements.

Note: the response is ordered by symbol name, not by the order you passed the symbols. Index it by `symbol`, never by request position.

Availability: in the default tool set; not usable as a sub-query of `batch` (heavyweight renderer).

`verify_claim`

Purpose. Verify a structured claim about a change by diffing two analyzed sessions (baseline → changed). Returns a verdict of `confirmed`, `refuted` or `indeterminate`, with a reason and evidence.

When to use it. To check a promise you or an agent made about a change - for example "this change adds no public API" - against the analyzed baseline and the analyzed result.

Parameter	Type	Required	Default	Meaning
`baselineSessionId`	string	yes	-	The baseline session (before the change).
`changedSessionId`	string	yes	-	The changed session (after the change).
`claimType`	string	yes	-	`no_new_api` or `symbol_removed_unused`.
`symbol`	string	no	`null`	Required for `symbol_removed_unused`: the symbol name the claim is about. A bare name or the qualified `Type.Member` form - the same strings `find_usages`, `impact_of_change` and `find_tests_for` resolve.

Claim types.

- `no_new_api` - no types or methods were added and no method signature changed between the two states. The check is overload-aware: an added overload and a signature change on any overload are detected.
- `symbol_removed_unused` - the named symbol was removed and had no usages in the baseline's static fan-in graph.

Note - the scope of `no_new_api`: it is about additions and signature changes, so a removal does not refute it; a change set that only deletes public API is legitimately `confirmed`. Removals are therefore never silent: the reason states the boundary, and every removal the same diff records is listed as evidence, prefixed with `-`. For the removal question, ask `symbol_removed_unused`.

Note: the tool is conservative in both directions - what the diff cannot prove is never falsely confirmed and never falsely refuted. `indeterminate` is returned, with the cause named, for a name that resolves to nothing; for a property, field, event or enum member (the diff models types and methods only); and for the simple name of a removed type that the diff had to render by its full name because it is ambiguous - pass that full name instead. A removed method whose only baseline callers called an interface or abstract member it implemented (`find_usages` ' `viaContract`) is `indeterminate` too, and the reason says whether that credit was exact. Baseline and changed must be two states: the same session twice (also an id and the solution path it was analyzed from), or two sessions of one solution with an equal file-set hash and an empty diff, answer `indeterminate`, and the reason says so and how to get two states (for example the old commit analyzed in a second worktree). An unsupported claim type also returns `indeterminate`.

For `symbol` with `symbol_removed_unused`: a namespace-qualified type name is accepted even when nothing was removed - this tool resolves it against the baseline's declared type identities, which the fan-in tools do not, and the answer then names the simple name to carry on with. A dotted name that no namespace declares stays unresolved rather than falling back to whatever type carries its last segment.

Response. `claim`, `verdict`, `reason`, `evidence` (the change list or the carriers the verdict rests on).

Availability: in the default tool set; not usable as a sub-query of `batch` (it needs two sessions).

`evaluate_change_set`

Purpose. An advisory policy gate: which code-quality, design-smell and layer-violation findings would this edit introduce? Ask before writing it. Findings are a delta against the current session, so pre-existing debt is never blamed. The change set is applied as an in-memory overlay and the amended solution is re-analyzed; nothing is written to disk and the session is untouched.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The baseline session id.
`changes`	array	yes	-	The proposed file changes (see below). At least one is required.
`policy`	string	no	`null` (= `block_on_critical`)	`block_on_critical`, `block_on_warning` or `advisory`. An unknown policy is rejected, not silently treated as `advisory`.
`dbPath`	string	no	`null`	The active QualityProfile (producer thresholds and toggles) and the configured default layer profile of that database. Omitted: the server's default configuration database, else the default profile.

The `changes` entries. Each entry needs `filePath` (absolute, matching the analyzed documents) plus one of two content forms:

- **Anchor patch (preferred):** `oldText` and `newText` - the snippet being replaced and what it becomes. This is the shape an editor takes, and the payload is the size of the change, not of the file: the full file is rebuilt from the analyzed document. `operation` may be omitted and defaults to `modify`.
- **Full text:** `newContent` - the complete new file content - with `operation` set to `modify` or `add`. A `delete` entry takes neither content field.

Anchor patch rules:

- `oldText` must occur exactly once in the file. A missing or ambiguous anchor is an error, never a silently different edit.
- Line endings are reconciled for you.
- An empty `newText` deletes the snippet.

The verdict. `pass` / `warn` / `block`, following the same policy table as `diff_review`: no introduced finding is `pass`; `advisory` is `warn` whenever something is introduced and never blocks; `block_on_warning` blocks on a warning or critical; `block_on_critical` blocks on a critical.

Response. `result`, `reason`, `policy` and `introduced`. Each introduced finding carries `id` (the producer-level insight id), `detail` (the display line, metric included), `severity` and `locator` (the structured symbol reference for that line, when the producer emitted one).

Note: the tool is advisory - an MCP tool cannot block. The real blocking gate is the CLI `--fail-on` option. `apply_change_set` is out of scope: this tool never applies your change.

Note: the tool reports an error for an unknown `operation`, a `filePath` that is not part of the analyzed solution, or a change that cannot be assigned to a target project.

Availability: in the default tool set; not usable as a sub-query of `batch` (it runs the heaviest re-analysis).

`compare_public_api`

Purpose. Compare the public API surface of two compiled .NET assemblies - a baseline `.dll` and a current `.dll` - and report the breaking changes as CPxxxx diagnostics: removed types (CP0001), removed members (CP0002), changed signatures, reduced visibility, and similar.

When to use it. Before shipping a library version, to check whether the new build breaks consumers of the old one.

Parameter	Type	Required	Default	Meaning
`baselineDllPath`	string	yes	-	Absolute path to the baseline (old) assembly `.dll`.
`currentDllPath`	string	yes	-	Absolute path to the current (new) assembly `.dll`.

No session and no solution is needed - both assemblies must be built.

Note: the tool runs the Microsoft `apicompat` command out of process. It must be installed as a local .NET tool in the repository the server runs in (`.config/dotnet-tools.json`); run `dotnet tool restore` once there. If a file is missing or the tool cannot run, the answer is an error, never a false "no breaks".

Response. `hasBreakingChanges`, `breakCount` and `breaks`, each break with `diagnosticId` and `message`.

Availability: not in the default tool set. Expose it by enabling the `api-surface` bundle in an MCP profile, or with `AICB_MCP_TOOLS`. Not usable as a sub-query of `batch`.

9.4 Insights

`list_insights`

Purpose. List the code-quality, async and design-smell insights for a session - long methods, fat interfaces, unused types, missing async suffixes, many-parameter methods, and more. Each insight is aggregated per producer, with the count of its detail lines; `get_insight` lists the lines, and `list_insights` lists them too when it is scoped to files.

When to use it. As the entry point of a quality review: it tells you which kinds of findings exist for this solution and how many of each. With `files` set to the files a change touches, it is the review sweep: the findings about the types those files declare, with their lines.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution`.
`dbPath`	string	no	`null`	The active QualityProfile (producer thresholds and toggles) and the configured default layer profile of that database. An explicit layer profile passed to `analyze_solution` wins over the database-configured one. Omitted: the server's default configuration database, else the default profile.
`files`	string array	no	`null`	C# files, each absolute or relative to the solution folder. The answer keeps the findings about the types these files declare and becomes an object (see "With `files`" below). Omitted or empty: the solution-wide array.

Response. An array of summaries, one per producer that found something:

Field	Meaning
`id`	The producer-level insight id, e.g. `quality-long-methods` - pass it to `get_insight`.
`featureId`	The feature identifier of the insight.
`title`	The producer's headline.

Field	Meaning
`severity`	`critical`, `warning`, `info` or `ok`.
`persistent`	`true` for reminder findings that are not removed by a dismissal.
`detailsCount`	Number of detail lines after filtering.
`remediationCostMinutes`	The producer's cost estimate, when it has one.
`suggestion`	A short, concrete fix sentence, when the producer has one.
`suppressed`	How many detail lines were removed by triage; `null` when nothing was filtered.

With `files`. The answer is an object `{files, notInSolution, insights}`. `files` lists the given files the solution holds, solution-relative; `notInSolution` the ones it does not (a deleted or non-C# file, a typo), as given - a path is never silently dropped. Each insight carries the fields above plus `details` and `detailLocators` (`{typeName, memberName}` per line): the lines about the types the files declare, up to 20, then a `(... and N more)` line; `detailsCount` counts the in-scope lines. `title`, `remediationCostMinutes` and `suppressed` stay whole-solution numbers, so `detailsCount` plus `suppressed` is not a split there. A line matches by the simple type name of its locator, so a same-named type elsewhere matches too; a line with no type locator is left out, and an insight about the solution as a whole (no lines) stays with `detailsCount` 0. A relative path is read against the solution folder, so pass absolute paths when your list comes from `git diff --name-only`, which prints paths relative to the repository root.

Note: this answer is triage-filtered on both axes. Findings the solution marked as intentional (a suppression, in the database or in the git-tracked `<Solution>.aicb.json` sidecar) and findings a user dismissed as seen (in the solution record, not the sidecar) are removed, and an insight whose findings are all gone does not appear at all.

Note: `suppressed` counts what both axes removed together, so it is not a count of sidecar entries. And `title` is producer text computed before filtering, so on a partially filtered insight it still names the pre-filter total - `detailsCount` plus `suppressed` are the real split.

Note: insights that depend on facts computed live during analysis (for example the unresolved-binding findings) are absent on a session recalled from a saved snapshot.

Note: the unused-type finding applies the type exemptions of `find_dead_code`, so a MEF part is not reported. The side-effect-concentration and security-sensitive-call findings judge methods only; the top-level-statement units that `calls_external` and `find_by_side_effects` list are not part of them.

Note: on a server without a config DB, and with no `dbPath`, the built-in default quality profile sets the thresholds and toggles. The answer then carries a `configDbNote` that says so, as a text block of its own after the array (with `files`, a member of the object); a staleness disclosure travels the same way after an array. With `aicb call`, a note that would follow the array goes to stderr, so stdout stays one JSON document.

Availability: in the default tool set; usable as a sub-query of `batch`.

`get_insight`

Purpose. Get the full detail of one insight by its id, including the per-item details list.

When to use it. After `list_insights` told you which producer found something, to see the individual findings.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session id returned by `analyze_solution`.
`insightId`	string	yes	-	The producer-level insight id, e.g. `quality-long-methods`. Discover the ids with `list_insights` or `list_insight_producers`.
`dbPath`	string	no	`null`	Same semantics as `list_insights`.

Parameter	Type	Required	Default	Meaning
`maxDetails`	integer	no	`null` (= 20)	Cap on the returned details list. Raise it (for example to 200) for a full per-item drill-down; a `(... and N more)` marker still discloses any remainder beyond the value. Applies only to this call; omitted or `0` or less keeps the default cap of 20.

Note: ids are case-sensitive. An unknown or misspelled id is rejected together with the known producer ids, so the error doubles as a did-you-mean list. A producer that is registered but found nothing in this session reports that explicitly - which is different from a typo, because `list_insights` lists only the producers that found something.

Response. The full insight: `id`, `featureId`, `title`, `description`, `actionLabel`, `severity`, `persistent`, `allowsPromptInput`, `secondaryActionLabel`, `details`, `remediationCostMinutes`, `suggestion`, `detailLocators` and `suppressedDetailCount`.

Note: triage-filtered like `list_insights`, on both axes. `suppressedDetailCount` reports the two axes added together, so it is not a count of sidecar entries. A producer silenced outright reports as "registered but 0 hits".

Note: without a config DB in force (no `dbPath`, and no default config DB on the server), a `configDbNote` member says that the built-in default quality profile judged.

Availability: in the default tool set; usable as a sub-query of `batch`.

`list_insight_producers`

Purpose. List the registered insight producers (id, name, group). This is a static catalog - no session is needed.

When to use it. To look up the exact `insightId` values for `get_insight`, including producers that found nothing in the current session and therefore do not appear in `list_insights`.

Parameters. None.

Response. An array of entries with `id`, `name` and `group`, ordered by group and then by name. `id` is the stable insight id you pass to `get_insight`; `name` is the producer's implementation name.

Note: producers that need the solution database - the public-API breaking-changes and LLM-findings producers - and the auto-compression producer are not available in the MCP server, so they do not appear in this catalog.

Availability: not in the default tool set; expose it with the `Full Select` profile (`aicb mcp --mcp-profile mcp-profile/full`) or `AICB_MCP_TOOLS`. Usable as a sub-query of `batch` (it takes no session).

10 Tool reference: snapshots, configuration, memory and wiring

This chapter documents the tools that persist and compare analyzed state, configure how a solution is analyzed, keep a cross-process memory of a codebase, browse the aicb database's master data, install the agent-side symbol guard, and run several read-only queries in a single call. The other tool families (symbol lookup, fan-in, tests, insights, markup, and so on) are documented in their own chapters.

Reading the parameter tables. `sessionId` is either an id returned by `analyze_solution` (or by a recall) or an absolute path to a `.sln`, `.slnx` or `.slnf` file. Passing a path analyzes that solution on first use and reuses the result afterwards, so a separate `analyze_solution` call is optional; see "Sessions and staleness". A tool marked as needing a *live* session refuses a session that was recalled from a database with `recall_codebase`, because the open analyzer workspace is gone.

`dbPath` **and the standard config database.** Several tools take an optional `dbPath` pointing at an aicb SQLite database. When a tool says "omit to use the server's standard config DB", the server falls back to the database it was started with - for the standalone `aicb mcp` command that is the same file the desktop GUI uses (`%APPDATA%/AIContextBuilder/user-data/aicb.acb`). A call without `dbPath` therefore reads and writes the database you actually work in; pass an explicit `dbPath` whenever the write should land elsewhere.

Pool membership. The shipped `Default` MCP profile exposes 54 of the 82 registered tools. `Full Select` exposes the 72-tool lean core. `compare_public_api` is reached through the opt-in `api-surface` bundle or an `AICB_MCP_TOOLS` override. Nine uncatalogued tools are available only through that override: `import_constellation`, `inspect_session`, `list_constellations`, `list_remembered`, `list_run_templates`, `list_sessions`, `recall_codebase`, `refresh_remembered`, `remember_codebase`. The variable (`all`, `lean`, or a comma-separated list such as `MemoryTools,DbEntityTools`) replaces the profile's tool set rather than adding to it, so name every group you need. The final index states Default membership, not reachability through `Full Select`.

10.1 Snapshots and comparison with a stored state

A snapshot is a persisted copy of the analyzed model of a solution inside an aicb database. `save_session` writes one; `compare_with_previous` and `diff_public_contract` compare the current session against one of the same solution. Snapshots are selected by age: `snapshotsBack` `0` is the newest saved snapshot, `1` the one before it, and so on.

`save_session`

Purpose. Persist the current session's analyzed model as a named manual snapshot in an aicb database, so that a later session can be compared against it.

When to use it. Before a refactoring or any larger change, so that "what changed since then" can be answered structurally later. The snapshot is a baseline of the analyzed model, not a backup of the source files.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session whose current state is snapshotted; live or recalled.
`name`	string	yes	-	A human-readable label, for example `before refactor`. An empty name is rejected.
`dbPath`	string?	no	`null`	The aicb database to persist into. Omit to use the server's standard config DB (the GUI's).

Response. - `snapshotId` - the id of the new snapshot. - `sessionId`, `solutionId` - the session and the solution record the snapshot belongs to. - `name` - the label you passed. - `lineNumbersAvailable` - whether the session carries line numbers. - `totalSnapshots` - how many snapshots this solution now has (manual and automatic).

Notes and limits. - The snapshot is stored as a *manual* snapshot. The automatic retention that caps the memory tools' snapshot history never deletes a manual snapshot, so a baseline stays until it is removed deliberately. - The snapshot keeps the line spans of the analysis, but a session recalled from it reports `lineNumbersAvailable: false`: its lines mean "as analyzed", not the current working tree. - Without an explicit `dbPath` the snapshot lands in the database the GUI uses.

`compare_with_previous`

Purpose. Diff the current session against a previously saved snapshot of the same solution - the newest by default, or the N-th previous via `snapshotsBack`.

When to use it. To answer "what changed since the last known-good state": which types were added or removed and, per changed type, which methods were added or removed and which signatures changed. The diff is name- and signature-based; method bodies are not compared.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The "after" side of the diff.
`dbPath`	string?	no	`null`	The aicb database holding the saved snapshots. Omit for the server's standard config DB.
`snapshotsBack`	int	no	`0`	`0` = newest saved snapshot, `1` = the one before it, etc. Must be `>= 0` and within the saved history.
`typeName`	string?	no	`null`	Restrict the diff to this type. Simple name, case-insensitive. Not a substring match: `MyApi` never matches `MyApiExtensions`. A simple name also matches the namespace-qualified form the diff uses when a simple name collides - the answer then spans every type of that name and `note` says which.
`methodName`	string?	no	`null`	Within that type, restrict to this method name (case-insensitive).

Response. - `sessionId` - the session that was compared. - `note` - present when the filter owes a disclosure, or when the baseline snapshot looks like the session's own state (see below); omitted otherwise. - `baselineSnapshotId`, `baselineName`, `baselineCreatedUtc` - which snapshot was used as the baseline. - `snapshotsBack`, `totalSnapshots` - the index used and the size of the history. - `diff` - `addedTypes`, `removedTypes`, `changedTypes`, plus `isIdentical` and `totalChangeCount`. Each changed type carries `typeName`, `addedMethods`, `removedMethods` and `changedMethodSignatures`; a signature entry gives `methodName`, `leftSignature`, `rightSignature` and the flags `returnTypeChanged`, `parametersChanged`, `accessibilityChanged`, `staticChanged`.

Notes and limits. - At least one earlier `save_session` for the same solution is required; otherwise the call is rejected with a message telling you to call `save_session` first. A `snapshotsBack` outside the saved history is rejected with the number of available snapshots. - The filter disclosure is the important property. When a filter empties a diff that did have changes, `note` says so and names the axis that missed, because `isIdentical: true` then describes the filtered view and not the solution. A changed type whose method lists the `methodName` filter emptied is dropped from the answer, so an empty view is never evidence that the method is unchanged. - On a diff that was already empty the filter owes no note: "your filter matched nothing" and "nothing changed" are the same state there, so a mistyped and a real name answer identically. Check the spelling with `find_symbol` rather than reading that agreement as confirmation. - A snapshot saved from the same source state as the session (the file-set hash is unchanged) and an empty diff get a note: the diff most likely compares that state with itself, so `isIdentical: true` is not evidence that a change kept the solution's shape. Save the snapshot before the change, edit, call `refresh_session`, then compare. When the snapshot with that hash was saved by another analyzer build, the note says so whether or not the diff is empty: a difference can come from the analyzer instead of the source.

`diff_public_contract`

Purpose. Report the breaking and additive changes to the public API surface of the current session compared with a saved snapshot of the same solution.

When to use it. Before releasing a library or any code other callers depend on: this answers "did I break a caller?" at the contract level rather than at the line level.

Not exposed by the `Default` profile - reachable under `Full Select` or via `AICB_MCP_TOOLS`.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The "after" side of the diff.
`dbPath`	string?	no	`null`	The aicb database holding the saved snapshots. Omit for the server's standard config DB.
`snapshotsBack`	int	no	`0`	`0` = newest saved snapshot, `1` = the one before it, etc.
`scope`	string?	no	`null`	Restrict the diff to a namespace prefix (case-insensitive).
`includeInternal`	bool	no	`false`	Also diff the internal family, not only the externally visible public/protected contract.

Response. - `sessionId`, `baselineSnapshotId`, `baselineName`, `baselineCreatedUtc`, `snapshotsBack`, `totalSnapshots` - the baseline metadata. - `scope`, `includeInternal` - the echo of what was compared. - `hasBreakingChanges`, `breakingCount`, `additiveCount` - true totals, even when a list below is truncated. - `removedTypes`, `removedMembers`, `changedMembers`, `addedTypes`, `addedMembers` - the five diff lists, each a capped envelope with `items`, `count`, `totalFound` and `truncated`.

Notes and limits. - Classification: a removed public type or member, or a changed member signature, is **breaking**; an added type or member is **additive**. Events count as members: a removed public event is breaking, an added one additive. - Both sides are projected through the same public-surface view, so the comparison is like for like. - The comparison is name- and signature-based; method bodies are not diffed. A change of a parameter *type* appears as a removal plus an addition. - At least one earlier `save_session` is required; a missing baseline is rejected with a message telling you to save one first.

10.2 Solution configuration (layers, exclusions, test detection)

Three per-solution settings decide how aicb reads your code: the **layer profile** (which namespace belongs to which architectural layer), the **namespace exclusion list** (which namespaces are not yours), and the **test-detection profile** (which projects are test projects and which attributes mark a test method). The four tools below form one guided flow: check the status, fetch the material, apply a proposal, and later check whether the configuration still fits the code. Only `apply_solution_config` writes.

`solution_config_status`

Purpose. Check whether the three axes have been initialized through the guided setup, and which layer profile, exclusion list and test profile are currently active.

When to use it. On first contact with a solution, before proposing a setup.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session of the solution to inspect.
`dbPath`	string?	no	`null`	The aicb config/master database. Omit to use the server's standard config DB (the GUI's).

Response. - `solutionId`, `solutionName`. - One slot per axis - `layerProfile`, `exclusionList`, `testProfile` - each with `initialized`, `initializedAt`, `activeId`, `activeName` and `source`. The test slot additionally carries `isCustom`.

Notes and limits. - Although the tool is observational, resolving an unknown solution ensures its row in the configuration database and can insert that row with the auto-initialization flags enabled. - `initialized` is an **explicit marker** stamped by `apply_solution_config`. It is not inferred from an active id, because an active profile can also be set by hand in the Settings panel. - `source` says where the axis comes from: `db` (a row in the config database, with its timestamp), `sidecar` (the git-tracked `<SolutionName>.aicb.json` next to the .sln, which is read at analyze time), or `none`. A repository with a valid sidecar and no database row therefore reports `initialized: true` with `source: "sidecar"`, not a misleading "not initialized". - The test slot resolves the effectively active test profile through the full chain (per-solution setting, then the global application setting, then the built-in default). `activeId` is `null` when the built-in default heuristic is in effect; `isCustom` tells you whether the resolved profile is a custom profile rather than a built-in preset. - If a slot is not initialized, offer `init_solution_config` followed by `apply_solution_config`.

`init_solution_config`

Purpose. Start the guided setup and return the raw material for a proposal. The tool writes nothing.

When to use it. After `solution_config_status` reports an axis as not initialized, and before `apply_solution_config`.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Must be a live session.

Response. - `solutionPath`. - `declaredNamespaces` - the solution's own namespaces, production-focused (test-project and framework/generated namespaces are dropped); the source for layer rules. Capped at 300 entries with `count`, `totalFound` and `truncated`. - `referencedNamespaces` - every namespace referenced through `using`, including namespaces owned by the solution itself. Capped at 300 entries. When proposing exclusion rules, use only its framework and third-party subset. - `testProjects` - the projects the default heuristic classifies as test projects. Capped at 300 entries. - `detectedTestAttributes` - the test-attribute markers that actually occur in the code, out of the built-in set (`Fact`, `Theory`, `Test`, `TestMethod`, `TestCase`). Not capped; the set is small by construction. - `proposalFormat` - a format hint that spells out the expected `proposalJson` and `testProposalJson` shapes.

Notes and limits. - Requires a live session: a session recalled from memory has no live workspace. The tool walks the already-warm session, so it triggers no re-analysis. - Budget for the answer rather than for the work: on a mid-sized solution the four lists arrive in one reply and can be several thousand tokens. Check the `truncated` flag before treating a namespace list as complete - otherwise your rules come from a sample. - This tool is not one of the read-only query tools that `batch` or `measure` can dispatch, so `measure` cannot price it.

`apply_solution_config`

Purpose. Apply a proposed layer, exclusion and test configuration: create a new custom layer profile, exclusion list and/or test profile (tagged with the source solution for provenance), activate them, and mark each touched slot as initialized.

When to use it. As the second half of the guided flow, using the material from `init_solution_config` or rules of your own.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The session of the solution to configure.
`dbPath`	string?	no	`null`	The config/master database to write into; created if missing. Omit to use the server's standard config DB (the GUI's).
`proposalJson`	string?	no	`null`	A JSON object with `layerRules` and/or `exclusions` (see below). Omit for a test-only apply.
`layerProfileName`	string?	no	`null`	Display name of the created layer profile. Omit for an auto-generated name.
`exclusionListName`	string?	no	`null`	Display name of the created exclusion list. Omit for an auto-generated name.
`layeringPolicy`	string?	no	`null` (`Advisory`)	`Advisory` or `Strict`.
`testProposalJson`	string?	no	`null`	A JSON object with `testProjectRules` and `testAttributeNames`; the rules match **project** names. Mutually exclusive with `testPreset`.
`testPreset`	string?	no	`null`	Shortcut: clone a built-in test profile (`xunit`, `nunit`, `mstest`, `default`) into a custom per-solution profile. Mutually exclusive with `testProposalJson`.
`testProfileName`	string?	no	`null`	Display name of the created test profile. Omit for an auto-generated name.

`proposalJson` shape:

```
{
  "layerRules": [ { "pattern": ".Application.", "matchType": "Contains", "layer": "Application" } ],
  "exclusions": [ { "pattern": "System.", "matchType": "StartsWith" } ]
}
```

`matchType` is `Contains`, `StartsWith`, `EndsWith` or `Exact`. Layer rules default to `Contains` and exclusions to `StartsWith` when `matchType` is omitted. An invalid `matchType` is rejected rather than silently defaulted - for a writing tool, a silent fallback would turn a mistyped `EndsWith` rule into the broader `Contains` rule.

`testProposalJson` shape:

```
{
  "testProjectRules": [ { "pattern": ".Tests", "matchType": "EndsWith" } ],
  "testAttributeNames": [ "Fact", "Theory" ]
}
```

Response. - `layerProfileInitialized`, `layerProfileId`, `layerProfileName`, `layerRuleCount`. - `exclusionListInitialized`, `exclusionListId`, `exclusionListName`, `exclusionRuleCount`. - `testProfileInitialized`, `testProfileId`, `testProfileName`, `testProjectRuleCount`. - `warnings` - non-fatal problems, including a failed sidecar write. - `sidecarPath` - the written sidecar file; omitted when the sidecar write failed.

Notes and limits. - At least one axis must be non-empty; empty arrays and slots are skipped, and a proposal that leaves all three axes empty is rejected with a message naming the accepted forms. - Auto-generated display names follow the pattern `LLM init (<solution name>) - layers`, `- exclusions` and `- tests`. - The tool writes to **two places at once**: the config database, and the git-tracked `<SolutionName>.aicb.json` sidecar next to the .sln. The returned `sidecarPath` comes with the explicit request to commit it, so the portable configuration travels with the repository. A later database-free analysis consumes its layer/exclusion fallback and analysis scope; the sidecar test definition is setup material rather than part of that resolution chain. This sidecar write is headless and does not ask for overwrite confirmation, unlike the GUI's export. - If the sidecar write fails (for example in a read-only directory), the database write is **not** rolled back; the failure is disclosed in `warnings` and `sidecarPath` is omitted. - The sidecar is rewritten as a whole, so its existing triage suppressions and the analysis-scope setting are read first and carried forward. Saving a configuration does not reset your triage. - An unknown `testPreset` is rejected with the list of known presets; passing `testProposalJson` and `testPreset` together is rejected as well. - Omit `dbPath` and the write lands in the database the GUI uses. If you keep more than one database, name the target explicitly.

`check_solution_config_drift`

Purpose. Check whether the active configuration still fits the code - has it drifted since it was set up?

When to use it. After the code has grown (new projects, new namespaces, new dependencies), or periodically as a health check.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	Must be a live session.
`dbPath`	string?	no	`null`	The aicb config/master database. If given, its active layer profile, exclusion list and test profile take precedence over the sidecar. Omit to fall back to the server's standard config DB, or - when the server has none - to the `.aicb.json` sidecar only.

Response. One axis object each for `layer`, `exclusion` and `test`, with: - `source` - where the checked configuration came from (`db`, `sidecar`, `built-in` or `none`). - `name` - its display name. - `status` - `stale`, `ok`, `not-configured` or `built-in-preset`. - `stale` - the boolean shorthand for `status == "stale"`. - `considered` - the size of the examined set (the denominator); `0` when the axis was not evaluated. - `sample` - the capped evidence list (up to 50 entries), whose `totalFound` is the full drift count. - `reason` - a sentence explaining the result.

The three signals.

Axis	Signal	Meaning
Layer	unmapped	Declared (own) namespaces that no rule of the active layer profile maps.
Exclusion	uncovered	External referenced namespaces (the solution's own are excluded from this check) that the active exclusion list does not cover.
Test	unmatched	Projects the canonical default heuristic classifies as test projects that the rules of the active custom test profile do not match - the profile may be too narrow.

Notes and limits. - The tool changes no configuration and creates no solution row. A `dbPath` that does not exist is refused with an error; an existing database of an older schema is upgraded on open (schema and built-in rows), as the server upgrades its own at startup. - Only a **custom** configuration is evaluated. A built-in preset (for example the Clean Architecture layer profile, the BCL exclusion default or the default test heuristic) and an empty or role-heuristic configuration are reported as `not-configured` or `built-in-preset` - a deliberate generic choice is not stale. - The layer and exclusion axes report `stale` from 3 drifted names on, the test axis when more than a quarter of the considered projects drifted; below that an axis reports `ok` with the evidence still listed. `sidecarProblem` appears when the `.aicb.json` could not be read. - A database-active id that no longer resolves (a deleted profile) falls back cleanly to the sidecar or to `none`. - Requires a live session.

10.3 Cross-process memory

The four memory tools persist an analyzed codebase in an aicb database and bring it back later - from the same process or a different one - without running the analyzer again. They are not part of any facet menu or bundle, so **no shipped profile exposes them**, and they are also not reachable through `batch` or `measure`. An operator exposes them with the `AICB_MCP_TOOLS` environment variable, for example `AICB_MCP_TOOLS=MemoryTools`.

`remember_codebase`

Purpose. Analyze a `.sln` and persist it as a snapshot in an aicb database so it can be recalled later, even from another process. Returns a **live** `sessionId` (line numbers intact) that you can use immediately.

When to use it. To make a codebase available to a later server process - for example a second harness or a nightly job - without paying for a fresh analysis there.

Parameter	Type	Required	Default	Meaning
`solutionPath`	string	yes	-	Absolute path to the `.sln` file to analyze and remember.
`dbPath`	string	yes	-	Absolute path to the aicb memory database to persist into. There is no server default; the database is created and migrated if it does not exist.
`layerProfile`	string?	no	`null`	Absolute path to a layer-mapping profile JSON. Omit for role-based heuristics.

Response. The session metadata: `sessionId`, `solutionPath`, `solutionName`, `projectCount`, `fileSetHash`, `layerProfile`, `lastAccessUtc`, `origin`, `lineNumbersAvailable` (and `analyzePreferredTfmOnly` when the sidecar asked for the preferred target framework only).

Notes and limits. - The persisted snapshot carries the same analysis scope as an `analyze_solution` run of the same solution: the namespace exclusions and the preferred-target-framework setting from the git-tracked `<SolutionName>.aicb.json` sidecar are applied. - Each call appends a new automatic snapshot. The automatic history per solution is capped at 10 entries, so older automatic snapshots are pruned; manual snapshots written by `save_session` are never pruned by this cap. - Remembering the same solution again after an edit re-analyzes it and writes a fresh snapshot, so the memory database does not keep serving a stale model from a warm cache.

`list_remembered`

Purpose. List the codebases remembered in an aicb database.

Parameter	Type	Required	Default	Meaning
`dbPath`	string	yes	-	Absolute path to the aicb memory database.

Response. A JSON array, one entry per remembered solution, ordered by solution path: - `solutionId`, `solutionPath`, `name`. - `lastSnapshotUtc` - the timestamp of the latest snapshot. - `fileSetHash` - the file-set hash of that snapshot. - `hasModel` - whether a usable model payload is present.

Notes and limits. Metadata only; no payload is loaded. `lastSnapshotUtc`, `fileSetHash` and `hasModel` come from the latest snapshot of each solution.

`recall_codebase`

Purpose. Recall a previously remembered codebase **without** running Roslyn - instant cross-process context.

When to use it. To start working on a codebase in a new process without paying for a fresh analysis, or to answer structural questions when the analyzer cannot run.

Parameter	Type	Required	Default	Meaning
`solutionPath`	string	yes	-	Absolute path to the `.sln` that was remembered.
`dbPath`	string	yes	-	Absolute path to the aicb memory database.

Response. `session` (the session metadata, including the new `sessionId`), `stale`, `lineNumbersAvailable` (always `false`), `reason`.

Notes and limits. - A recalled model has **no line numbers**, and line-number-dependent insights are degraded. Call `refresh_remembered` for a live re-analysis. - `stale` is `true` when the files changed since the snapshot, or the payload schema or the analyzer identity differs. For a stale result, `reason` lists these possible causes in one fixed sentence; it does not identify which one occurred. - The recalled model is served **even when it is stale** - that is the point of instant cross-process context. Treat a stale answer as a hint and refresh before relying on it. - The `.cshtml` view census, the generated-code censuses and the top-level-statement facts travel with the snapshot, so the recalled session's answers carry the same notes as a live one. Where the snapshot was stored before such a fact was recorded, the affected answers say `VIEWS NOT KNOWN`, `GENERATED CODE NOT KNOWN`, `TOP-LEVEL CREATIONS NOT KNOWN` or `TOP-LEVEL EFFECTS NOT KNOWN` instead (see "Recall safety" under "Recurring conventions across the catalog"); `refresh_remembered` clears them. - The lookup is read-only: a solution that was never remembered is rejected with a message telling you to call `remember_codebase` first.

`refresh_remembered`

Purpose. Re-analyze a recalled session live (acquiring a fresh analyzer workspace), restoring line numbers and full insights, and re-persist the snapshot.

When to use it. After `recall_codebase` has served its instant answer and you need line-accurate data or the full insight set.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The recalled session id from `recall_codebase`.
`dbPath`	string	yes	-	Absolute path to the aicb memory database to re-persist into.

Response. `session` (a fresh **live** session, including its new `sessionId`), `lineNumbersAvailable` (`true`), `reason`.

Notes and limits. - The old recalled session is evicted; use the returned id from now on. - The live re-analysis applies the same sidecar settings as a fresh `analyze_solution`, so it yields the same facts and the same project inventory. - The snapshot is re-persisted (and the automatic-history cap applied) before the old session is dropped, so a failure leaves the old session intact.

10.4 aicb database entities

These three tools browse and import the master data of an aicb database rather than the analyzed code. They are not part of any facet menu or bundle, so no shipped profile exposes them; expose them with `AICB_MCP_TOOLS` (for example `AICB_MCP_TOOLS=DbEntityTools`).

`list_run_templates`

Purpose. List the run templates available in an aicb database. Built-in run templates are always returned; passing an absolute `dbPath` additionally includes the custom ones from that database.

Parameter	Type	Required	Default	Meaning
`dbPath`	string?	no	`null`	An absolute path to an aicb database. Omit to use the server's standard config DB (the GUI's) if it resolved one, otherwise only the built-ins are listed.

Response. A JSON array, ordered by id, each item with `id`, `name`, `isBuiltIn`, `isOverridden`, `isHidden` and `builtInsOnly`.

Notes and limits. `builtInsOnly` is `true` when only the built-in templates were listed, i.e. no database was used. An explicit `dbPath` that does not exist is rejected (it is not created), and a directory is rejected as well.

`list_constellations`

Purpose. The same listing for constellations (context templates).

Parameter	Type	Required	Default	Meaning
`dbPath`	string?	no	`null`	As for `list_run_templates`.

Response. The same item shape as `list_run_templates`, ordered by id.

`import_constellation`

Purpose. Import a constellation JSON file into an aicb database, following the read → preview → apply sequence.

When to use it. To move a set of master-data entities (prompts, context templates, run templates, layer profiles, exclusion lists, test profiles, MCP profiles, quality profiles and more, plus application-settings keys) from one database or machine to another.

Parameter	Type	Required	Default	Meaning
`filePath`	string	yes	-	Absolute path to the constellation `.json` file to import. Must exist.
`dbPath`	string	yes	-	Absolute path to the target database. Created if it does not exist; a directory is rejected.
`mode`	string	no	`"SkipExisting"`	`SkipExisting` keeps existing ids; `Replace` overwrites them. Any other value is rejected.
`previewOnly`	bool	no	`false`	When `true`, nothing is written and the per-section counts are returned instead.

Response (preview, `previewOnly: true`). - `name` - the constellation's name. - `applied` - always `false` on this path. - `sections` - one entry per section, with `entityType`, `newItemCount`, `conflictCount`, `unchangedCount` and `skippedBuiltInCount`. - `appSettingsKeys` - the application-settings keys the file carries.

Response (apply). - `name`, `dbPath`, `applied` (`true`). - `appliedItemCount`, `skippedItemCount`, `appSettingsKeysApplied`. - `warnings` - the non-fatal per-section errors of the apply.

Notes and limits. - The file is read and validated first, then previewed, then applied. A read or preview error aborts the call before anything is written. - Entries marked as built-in in the import file are always skipped. A custom entry whose id collides with an installed built-in is different: `Replace` can overwrite that row, while `SkipExisting` preserves it. Preview carefully before replacing. - An unknown section type is reported as an error rather than silently ignored. - Use `previewOnly` first when importing into a database you care about: it tells you exactly how many items are new, conflicting or unchanged.

10.5 Agent wiring

`install_agent_hooks`

Purpose. Install the aicb symbol guard into this project for an agent harness, so that a C#-symbol question is **refused** on a text search and redirected to the tool that answers it (with the two exits described under "Two ways past the refusal").

When to use it. When your client offers it, or when you want the guard without re-running the full `aicb init`.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	A session id, or the absolute path to a `.sln` / `.slnx` / `.slnf` file (self-init).
`harness`	string?	no	`null`	Which harness to install for: `claude-code`, `codex` or `opencode`. Omit to install for the harness the calling client belongs to.
`force`	bool	no	`false`	Overwrite an existing guard script and rewrite an existing wiring entry.

Response. - `harness` - the harness id; `harnessName` - its display name (`Claude Code`, `Codex CLI`, `OpenCode`). - `projectDirectory` - the directory the artifacts were written into. - `succeeded` - `false` when any artifact was refused. - `artefacts` - one entry per file touched or declined, each with `path`, `outcome` and `detail`. `outcome` is one of `Created`, `Updated`, `Unchanged`, `Refused`. - `nextStep` - the follow-up instruction (see below).

Notes and limits. - What is written: the guard script and the harness's wiring entry - `.claude/settings.json`, `.codex/hooks.json` or `opencode.json` - and nothing else. No `.mcp.json` and no skills, and only inside the directory of the solution the session refers to. For OpenCode the guard is delivered as a plugin under `.opencode/plugins/` plus the `opencode.json` entry. - An existing guard or wiring is kept unless `force=true`, so a re-run cannot silently discard your own edits - an edited exemption list is the obvious thing to lose here. - The tool installs for the calling harness by default. If the client announces no name aicb can map to a harness, or you name an unknown one, the call is refused with the list of known harnesses rather than writing a configuration nothing would read. - After the call: restart or reconnect the MCP client so it loads the new wiring, then call `refresh_session` so the session stops reporting the guard as missing. The guard runs in the client, which reads its configuration at startup, so it is not active before the reconnect. - This tool belongs to the always-available navigation core: the offer a session makes to install the guard must be acceptable in the `Default` profile, otherwise accepting it would fail.

10.6 Multi-dispatch

`batch` and `measure` run several read-only queries in one round trip. They share one call shape and one registry of allowed tools; the difference is what they do with the answers - `batch` returns them, `measure` returns their size.

`batch`

Purpose. Run several read-only query tools in one round trip instead of calling them as separate turns. All sub-queries reuse one session, so a self-initializing `.sln` path is analyzed only once.

When to use it. Whenever several answers about the same symbol are needed at once - for example `find_usages`, `get_type_hierarchy` and `find_tests_for` - or `list_insights` followed by `get_insight`.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The shared session, or an absolute `.sln` path for self-init. Shared by all sub-queries - do not repeat it inside the queries.
`queries`	`BatchQuery[]`	yes	-	An array of `{ tool, args }`: `tool` is a read-only tool name, `args` is that tool's own arguments as a JSON object **without** the `sessionId`. `args` may be omitted for a tool that needs no arguments. Maximum 16.

Example:

```
{
  "sessionId": "<session id or absolute .sln path>",
  "queries": [
    { "tool": "find_usages", "args": { "symbol": "OrderService" } },
    { "tool": "get_type_hierarchy", "args": { "typeName": "OrderService" } },
    { "tool": "find_tests_for", "args": { "symbol": "OrderService" } }
  ]
}
```

Response. `{ results: [ { tool, ok, result | error, note } ], count, okCount }`. - Each sub-query is isolated: a failing one reports `ok: false` plus an `error`, while the others still return normally. One bad query never fails the batch. - A JSON-returning tool's result is nested as a JSON object; a Markdown-returning tool's result (for example `get_context` or `explain_symbol`) is the Markdown string. - `note` carries the sub-query's ignored-arguments disclosure: an argument name the tool does not declare was dropped, and the note names it (up to eight names, then a count) and lists the tool's real parameters. The field is omitted when there is nothing to disclose.

Notes and limits. - A sub-query has exactly two fields, `tool` and `args`; there is no caller-assigned id. Because several sub-queries may name the same tool, error and omission messages carry a short echo of the identifying arguments (for example `symbol=OrderService`) so you can tell which query they refer to. The echo is part of the message text, not a separate field. - The response budget is about 9,000 tokens. A result larger than that, or one that would push the running total over it, is replaced by `ok: false` with its measured size and a pointer to call that tool directly (slice tools accept a smaller `budget`). The work has already been done; only the response is trimmed. In practice this only affects the slice tools (`get_context`, `explain_symbol`, `pack_for_task`). - The maximum is 16 sub-queries per call; a larger array fails the whole call with a message telling you to split it. An empty `queries` array fails the whole call as well. A bad `sessionId` fails the whole call once, up front. A cancellation aborts the whole call and is never reported as a sub-query error. - Sub-queries run sequentially, in the order you list them. - **Only read-only query tools can be dispatched.** The registry holds 48 tools: `architecture_overview`, `assert_absence`, `call_graph`, `calls_external`, `check_doc_drift`, `check_pattern_drift`, `confidence_map`, `coverage_gaps`, `describe_api_surface`, `detect_circular_dependencies`, `explain_symbol`, `find_binding_usages`, `find_by_attribute`, `find_by_code_traits`, `find_by_complexity_and_coverage`, `find_by_concurrency_risk`, `find_by_event_subscription`, `find_by_resource_leak`, `find_by_returns_semantic`, `find_by_semantics`, `find_by_side_effects`, `find_dead_code`, `find_god_objects`, `find_implementations`, `find_overrides`, `find_production_dead`, `find_resource_usages`, `find_structural_twins`, `find_symbol`, `find_tests_for`, `find_unresolved_bindings`, `find_usages`, `get_context`, `get_diagnostics`, `get_insight`, `get_type_hierarchy`, `impact_of_change`, `instantiation_sites`, `lifecycle_of`, `list_insight_producers`, `list_insights`, `pack_for_task`, `resolve_injection`, `solution_metrics`, `symbol_metrics`, `symbol_signature`, `trace_flow`, `type_dependency_path`. - A tool that is read-only and in the registry but outside the **active profile's pool** is refused per item with its own message: the active pool governs `batch` exactly as it governs a direct call. Under the `Default` profile this affects several registry entries, so check `list_skills` for the active pool before assuming a name is callable. - The refusal message for a name outside the registry names the four kinds of reason a tool can be excluded, and the names in each bracket are **examples, not the whole group** - a tool's absence from them does not mean it writes anything. The kinds are: (1) it mutates session state or writes (`analyze_solution`, `refresh_session`, `apply_solution_config`, `save_session`, `remember_codebase`, `export_markdown`, `install_agent_hooks`) - the security boundary; (2) it needs a second state, another session or a stored snapshot (`semantic_diff`, `diff_review`, `verify_claim`, `compare_with_previous`); (3) `batch` and `measure` themselves (recursion); (4) it is read-only but a poor batch member (`evaluate_change_set`, `init_solution_config`, `prepare_task`,

`review_context`) - heavyweight walks, renderers and multi-file payloads whose answers would routinely exceed the response budget. - The trailing `Available:` line of a refusal lists every tool the dispatch registry knows, which can include tools the active profile does not expose. Treat it as a spelling aid, not as the active pool. - A tool used only through `batch` still appears in the usage statistics: the batch call records the tools it dispatched, so such a tool is visible even though it has no call row of its own. - `batch` is part of the navigation core and therefore always available.

`measure`

Purpose. Report how big a read-only query's answer would be - the exact token count - **without** returning the answer.

When to use it. Before pulling something expensive: to decide whether a call is worth making, which `budget` to pass a slice tool (`get_context`, `explain_symbol`, `pack_for_task`), or whether to narrow `scope` first.

Parameter	Type	Required	Default	Meaning
`sessionId`	string	yes	-	The shared session, or an absolute `.sln` path for self-init. Shared by all queries.
`queries`	`BatchQuery[]`	yes	-	The read-only queries to measure, exactly as for `batch`: `{ tool, args }`, `args` without the `sessionId`. Maximum 16.

Response. `{ measurements: [ ... ], count, okCount, totalTokens }`. - Each measurement carries `tool`, `ok` and, for a successful query, `tokens`, `chars`, `returned`, `totalFound`, `truncated` and `budgetNote`; a failed query carries `error` instead and no size fields. - `tokens` is the real tokenizer count (cl100k_base) of the exact text the tool would return. - `returned`, `totalFound` and `truncated` are passed through when the measured tool answers with a capped envelope, so you also see whether you would be seeing everything. For a tool that answers in Markdown (no envelope) they are omitted rather than reported as zero. - `budgetNote` carries the slice tools' budget-pruning disclosure verbatim: if it says types were dropped, the answer you are pricing is already cut - an axis you asked for can be missing entirely - and a larger `budget` is what buys it back. - `totalTokens` is the summed cost of all measured queries. - `note` carries the same ignored-arguments disclosure as in `batch`.

Notes and limits. - `measure` **does not make the query cheaper to run.** The server does the full work either way; it makes the answer cheap to *inspect* - you pay about 50 tokens instead of the answer. So measure to decide, then call once; do not measure every call by reflex. - `measure` dispatches the same read-only tools as `batch` and refuses the same ones for the same four reasons; the refusal message names which reason applies. - The one tool whose "call it directly" advice would be wrong is `export_markdown`: for it, `measure` advises giving the tool an `outputPath` instead. It then renders once, writes the file and returns the exact character count - cheaper than measuring, and you keep the render. - `measure` calls are recorded as ordinary calls; the tools they priced are not added to the usage report's sub-query tally. - `measure` is part of the navigation core and therefore always available.

10.7 Alphabetical index of all tools

The table lists every tool the server registers, the group it belongs to, and whether the shipped `Default` MCP profile exposes it. A `no` does not mean the tool is unavailable, but the route differs: facet-menu tools are reachable through `Full Select`; `compare_public_api` needs the `api-surface` bundle or `AICB_MCP_TOOLS`; and the nine uncatalogued tools listed above require `AICB_MCP_TOOLS`.

Tool	Group	In Default profile
`analyze_solution`	Session and analysis	yes
`apply_solution_config`	Solution configuration	yes
`architecture_overview`	Orientation and server introspection	yes
`assert_absence`	Tests and coverage	yes
`batch`	Multi-dispatch	yes

Tool	Group	In Default profile
`call_graph`	Fan-in, impact and paths	yes
`calls_external`	Fact queries	yes
`check_doc_drift`	Patterns and documentation drift	no
`check_pattern_drift`	Patterns and documentation drift	no
`check_solution_config_drift`	Solution configuration	yes
`compare_public_api`	Diff, review and change gates	no
`compare_with_previous`	Snapshots and comparison	yes
`confidence_map`	Fact queries	no
`coverage_gaps`	Tests and coverage	yes
`describe_api_surface`	Orientation and server introspection	yes
`detect_circular_dependencies`	Dead code, cycles and metrics	yes
`diff_public_contract`	Snapshots and comparison	no
`diff_review`	Diff, review and change gates	no
`docs`	Orientation and server introspection	yes
`evaluate_change_set`	Diff, review and change gates	yes
`explain_symbol`	Symbol lookup and signatures	yes
`export_markdown`	Packing and exporting context	yes
`find_binding_usages`	Markup (XAML/AXAML)	yes
`find_by_attribute`	Fact queries	yes
`find_by_code_traits`	Fact queries	no
`find_by_complexity_and_coverage`	Tests and coverage	no
`find_by_concurrency_risk`	Fact queries	no
`find_by_event_subscription`	Fact queries	yes
`find_by_resource_leak`	Fact queries	no
`find_by_returns_semantic`	Fact queries	no
`find_by_semantics`	Fact queries	yes
`find_by_side_effects`	Fact queries	yes
`find_dead_code`	Dead code, cycles and metrics	yes
`find_god_objects`	Dead code, cycles and metrics	no
`find_implementations`	Hierarchy, implementations and lifecycle	yes
`find_overrides`	Hierarchy, implementations and lifecycle	yes
`find_production_dead`	Dead code, cycles and metrics	no
`find_resource_usages`	Markup (XAML/AXAML)	yes

Tool	Group	In Default profile
`find_structural_twins`	Patterns and documentation drift	no
`find_symbol`	Symbol lookup and signatures	yes
`find_tests_for`	Tests and coverage	yes
`find_unresolved_bindings`	Markup (XAML/AXAML)	yes
`find_usages`	Fan-in, impact and paths	yes
`get_context`	Symbol lookup and signatures	yes
`get_diagnostics`	Session and analysis	yes
`get_insight`	Insights	yes
`get_type_hierarchy`	Hierarchy, implementations and lifecycle	yes
`impact_of_change`	Fan-in, impact and paths	yes
`import_constellation`	aicb database entities	no
`init_solution_config`	Solution configuration	yes
`inspect_session`	Session and analysis	no
`install_agent_hooks`	Agent wiring	yes
`instantiation_sites`	Fan-in, impact and paths	yes
`lifecycle_of`	Hierarchy, implementations and lifecycle	no
`list_constellations`	aicb database entities	no
`list_insight_producers`	Insights	no
`list_insights`	Insights	yes
`list_mcp_profiles`	Orientation and server introspection	yes
`list_remembered`	Cross-process memory	no
`list_run_templates`	aicb database entities	no
`list_sessions`	Session and analysis	no
`list_skills`	Orientation and server introspection	yes
`measure`	Multi-dispatch	yes
`pack_for_task`	Packing and exporting context	yes
`prepare_task`	Packing and exporting context	yes
`recall_codebase`	Cross-process memory	no
`refresh_remembered`	Cross-process memory	no
`refresh_session`	Session and analysis	yes
`remember_codebase`	Cross-process memory	no
`resolve_injection`	Dependency injection	yes
`review_context`	Diff, review and change gates	yes

Tool	Group	In Default profile
`save_session`	Snapshots and comparison	yes
`semantic_diff`	Diff, review and change gates	no
`server_info`	Orientation and server introspection	yes
`solution_config_status`	Solution configuration	yes
`solution_metrics`	Dead code, cycles and metrics	yes
`symbol_metrics`	Dead code, cycles and metrics	yes
`symbol_signature`	Symbol lookup and signatures	yes
`trace_flow`	Fan-in, impact and paths	no
`type_dependency_path`	Fan-in, impact and paths	no
`usage_report`	Orientation and server introspection	yes
`verify_claim`	Diff, review and change gates	yes

11 Configuration, drift and usage recording

This chapter covers the surfaces that surround the MCP tools: how an `aicb mcp` process is configured, how you find out whether the running server still matches that configuration, what the server records about the calls it serves, and how to read or delete that record. It closes with the two surfaces a client can use besides `tools/call`: the agent-wiring tool and the MCP resources.

11.1 Server configuration

An MCP server process takes its settings from three sources, in this order of precedence:

built-in defaults < `aicb.mcp.json` < environment variables

The cascade applies to both configurable axes: which tools the server exposes, and how long it keeps analyzed solutions in memory.

The configuration file `aicb.mcp.json`

The file lives in your user configuration directory: `<ApplicationData>/AIContextBuilder/aicb.mcp.json` - on Windows that is `%APPDATA%\AIContextBuilder\aicb.mcp.json`. It is optional and **read-only for the server**: you edit it by hand, and the server never writes it.

```
{
  "toolSpec": "lean",
  "sessionCache": { "ttlMinutes": 90, "maxSessions": 8, "sweepMinutes": 5 }
}
```

Field	Values	Default	Effect
`toolSpec`	`"lean"`, `"all"`, or a comma-separated list of tool-group names	not set - the active profile's pool applies	Which tools the server exposes
`sessionCache.ttlMinutes`	positive integer	`90`	Session lifetime in minutes, counted from the last access
`sessionCache.maxSessions`	positive integer	`8`	Maximum number of analyzed solutions held at the same time
`sessionCache.sweepMinutes`	positive integer	`5`	Interval of the background sweep that frees idle sessions, in minutes

The session cache holds the analyzed solutions a server process has opened. Each held session pins an MSBuild workspace and the analyzed graph, so the cache is memory-intensive: at `maxSessions` the least recently used session is evicted, and the background sweep releases sessions that have expired even when no tool is called.

About the file's behavior:

- **Unset fields keep their default.** An invalid or non-positive value is ignored, and the field stays at its default.
- The file is hand-written-friendly: keys are case-insensitive, and both comments and trailing commas are tolerated.
- **Missing, unreadable or broken means "no override".** The server then starts on the built-in defaults and writes a warning to its error stream. That warning is the only difference between a broken file and an absent one, so if you believe the file is in effect but nothing changes, check the server log of your client - a file the server could not parse looks exactly like a file that is not there.

The `toolSpec` values in detail:

Value	Meaning
`"lean"`	The complete 72-tool lean core, the same set as `Full Select`
`"all"`	Every tool this binary registers, including the long-tail tools that no built-in profile routes to
comma-separated tool-group names	Only the tools of the named groups, for example `DbEntityTools,MemoryTools`

An explicit spec - from the file or from `AICB_MCP_TOOLS` - makes the exposed set **static**: it no longer follows the active MCP profile. Without an explicit spec, the active profile is resolved when the server starts; profile edits require a restart.

A `methods:`-prefixed, comma-separated list of tool function names (for example `methods:find_usages,get_context`) is also accepted by the same setting, if you want to select individual tools rather than whole groups.

Environment variables

Every environment variable the MCP server reads, with its default:

Variable	Default	Effect
`AICB_MCP_TOOLS`	-	Tool-set spec: `all`, `lean`, or a comma-separated list of tool-group names. Overrides `toolSpec` from the file and makes the exposed set static
`AICB_MCP_SESSION`	-	Session cache as a comma-separated `key=value` list, for example `maxSessions=4,ttlMinutes=120`. Keys: `ttlMinutes`, `maxSessions`, `sweepMinutes`
`AICB_MCP_AUTO_REFRESH`	the active profile's mode (built-in profiles: `reactive`; without a config DB: `off`)	`off`, `reactive` or `proactive`; overrides the profile's mode for this process only
`AICB_MCP_DEBOUNCE_MS`	`15000`	Quiet window of the file watcher, in milliseconds
`AICB_MCP_STALENESS_WINDOW_MS`	`5000`	Throttle window of the staleness check, in milliseconds
`AICB_MCP_LOAD_WAIT_MS`	`45000`	How long a tool call waits at the load gate before it is told what is currently loading, in milliseconds
`AICB_MCP_ERROR_TEXT`	on	`off` records only the exception **type**, no message (see "Privacy and what leaves your machine")
`AICB_DESIGN_TIME_BUILD_CACHE`	on at `%LOCALAPPDATA%\AIContextBuilder\design-time-build-cache`	`0`, `off` or `false` disables the persisted design-time-build cache; an absolute path relocates it
`AICB_ANALYZE_PROFILE_PATH`	-	A path arms the opt-in analysis wall-clock profiler and selects its report file
`AICB_ANALYZE_PROFILE_BIND_FIRST`	off	With the profiler armed, `1`, `true`, `yes` or `y` forces full semantic binding before the document loop

Variable	Default	Effect
`AICB_ANALYZE_PREFETCH_DEPTH`	half the logical processors, clamped to 1-16	Non-negative number of compilation builds to keep ahead; `0` disables prefetch and values above 16 are capped
`AICB_ANALYZE_NULLABLE_FLOW`	off	`1`, `true`, `yes` or `y` retains nullable-flow analysis in semantic models

Details that matter in practice:

- `AICB_MCP_SESSION` accepts the three keys above in any order; an unknown key or a non-positive value is ignored and the corresponding field keeps its current value (the file's value if set, otherwise the default).
- `AICB_MCP_AUTO_REFRESH` also accepts the spellings `0`, `false`, `no` for `off` and `1`, `on`, `true`, `yes` for `proactive`. An unrecognized value is ignored rather than treated as `off`, so a typo can neither switch the mode on nor silently undo a configured mode. The modes are: `off` only discloses a drifted session in the answer; `reactive` repairs it before the tool answers; `proactive` additionally watches the solution folder for file changes. The mode is read once at server start.
- The three timing variables are positive integers in milliseconds. Zero, negative and unparsable values fall back to the built-in value; there is no upper bound. A zero debounce or a zero throttle would fire work on every keystroke or every call, which is why zero is refused rather than honored.
- Environment variables are per process: set them where your client launches the server, not in a shell you use for something else.

Starting the server with a specific database or profile

```
aicb mcp --db-path "D:\work\aicb.acb"
aicb mcp --mcp-profile mcp-profile/full
```

- `--db-path` names the configuration database for this process. An explicit path may name any existing SQLite file regardless of extension (empty paths and directories are rejected). Only the application-resolved standard database path is subject to the `.acb` extension check.
- Without `--db-path`, the server resolves the standard configuration database (the same one the desktop application uses, see below) and creates it on first start. If that resolution fails, the server starts **without a database** and says so on its error stream.
- `--mcp-profile` pins which MCP profile this process serves, for this process only. If the pin cannot be applied - no config DB, unknown id, failed migration - the server **refuses to start** instead of silently serving a different profile.

Which database the server uses

The server has one process-wide default database, resolved once at startup:

explicit `dbPath` argument > process default > no database

The process default is the DB given with `--db-path`, or the standard configuration database: `%APPDATA%\AIContextBuilder\user-data\aicb.acb`. The `dbPath` parameter of the following tools is optional; when you omit it, the process default applies:

`analyze_solution`, `list_insights`, `get_insight`, `solution_metrics`, `evaluate_change_set`, `diff_review`, `save_session`, `compare_with_previous`, `diff_public_contract`, `solution_config_status`, `apply_solution_config`, `check_solution_config_drift`, `list_run_templates`, `list_constellations`.

`semantic_diff` and `init_solution_config` declare no `dbPath` parameter. `import_constellation` requires an explicit target `dbPath`; it never falls back to the process default.

This is what makes the desktop application's settings apply automatically when an agent works on the same solution: the per-solution namespace exclusions, test detection and layer profile, the active quality profile, and your custom entities are all read from the same database, with no path to pass.

A server without a usable database is not broken, but it is **profile-blind**: it falls back to the built-in default profile's pool, and tools that genuinely need a database answer with `dbPath is required (...)` naming what the path was for. `usage_report` answers `available: false` in that state. The insight tools say so in their answer: `configDbNote` names the built-in default quality profile as the one that judged.

The per-solution sidecar

Alongside the database, a solution can carry its own configuration as a git-tracked file next to the `.sln`: `<SolutionName>.aicb.json`. Of the guided setup tools, only `apply_solution_config` persists the proposed configuration into the database and sidecar; `solution_config_status` reads status (while ensuring the solution record exists) and `init_solution_config` only gathers proposal material. `apply_solution_config` returns the written path as `sidecarPath`; commit it to version the portable configuration. The file can contain all three profile definitions, but not every headless resolution path consumes every axis from it.

For the MCP analysis itself, resolution is per axis:

Axis	MCP analysis resolution
Layer profile	explicit profile argument where offered > database (per-solution, then global) > sidecar > role-based heuristic
Namespace exclusions	database, as soon as a usable database exists > sidecar > none
Test profile	database (per-solution, then global) > built-in default; the sidecar's test definition is not in this chain
Analysis scope	sidecar
Design-time build platform	sidecar key `designTimeBuildPlatform` (written by hand; `apply_solution_config` keeps it when it rewrites the file)
Suppressions	not part of the semantic analysis configuration; suppression-aware reading tools combine their documented database and sidecar sources
Auto-init flags	reported as setup state where applicable, not evaluated by headless analysis

A repository with only a sidecar can therefore supply layer rules, exclusions and analysis scope to a database-free headless analysis, but it is not accurate to call every axis fully configured from that file alone.

`solution_config_status` reports, per axis, which source is currently active: `db`, `sidecar` or `none`.

Note: a session keeps the configuration it was analyzed with. `refresh_session` re-analyzes the source with that same configuration - it deliberately does not re-read the sidecar, because a silent configuration switch under an unchanged session id would be invisible. If you have edited the sidecar and want the new values to apply, call `analyze_solution` again (or restart the server or wait for the cached session to expire). Passing the same `.sln` path to another tool reuses a non-expired cached session and does not re-read the sidecar.

The design-time build platform is the one exception, because it is a property of the build rather than of the analysis: `"designTimeBuildPlatform": "x64"` names a platform as the solution's configuration manager does (`x64`, `ARM64`, `Any CPU`), and every full reload reads it afresh - `refresh_session` with `force: true` applies an edit, while an unforced refresh after an edit of the sidecar alone keeps the platform the session was loaded at. A value the solution file does not name, or one under which the solution maps its projects to several platforms, is refused and the automatic choice applies; a blank value or one that is not a JSON string makes the whole file malformed, like any other broken axis. `get_diagnostics` reports the platform that ran and where it came from.

What the desktop application writes and what the server reads

Setting	Stored in	When the MCP server reads it
Per-solution configuration (layer, exclusions, test detection)	config DB and the git-tracked `.aicb.json` sidecar	at analysis time, according to the per-axis matrix above; layer/exclusion sidecar fallback needs no `dbPath`, while headless test detection does not consume the sidecar test axis
Active MCP profile (tools, instructions, token budget)	config DB	once at server start: tool list, callability, instructions and auto-refresh mode
Session cache (TTL, max sessions, sweep)	`aicb.mcp.json` / `AICB_MCP_SESSION` only - there is no editor in the desktop application	at server start
Tool-set spec (`lean` / `all` / list)	`aicb.mcp.json` / `AICB_MCP_TOOLS` only	at server start (static once set)
Token budget of the active profile	config DB	per call, by every tool that renders a document
LLM settings (model profiles, retry)	config DB	not at all - the MCP server makes no LLM calls

Worth knowing: an edit to the active profile in the desktop application does not reach a running server - neither its tool list nor its instructions (sent once when the client connected) nor its auto-refresh mode (fixed at start). `server_info` reports that state as CONFIG DRIFT until you restart the server - see the next section.

11.2 Telling whether the server still matches your configuration

`server_info` is more than a reachability check. It answers the question an MCP client otherwise cannot ask: *does this answer describe my current state?* It needs no session, so it is safe to call at any time.

The answer is assembled from four parts:

```
aicb MCP server (AIContextBuilder) v<version> (commit <sha>) - Roslyn-based .NET context generator. (c) Gregor
Dadera - free for private, hobby and educational use and qualifying organizations; see LICENSE.txt.
Config DB schema: user_version=<n> (<path to the config DB>).
<analyzer drift line>

<version drift block>

⚠ CONFIG DRIFT: <message>
```

- The **version** is followed immediately by the **build commit**. That placement is deliberate: the failure it exists for is a version number that reads correctly while the binary is not the one you think. A version number can be identical across two different builds - a parallel build that packed the same number first wins, and a package update then skips the newer one as "already installed". The commit is the only field that shows this; compare it with `git rev-parse HEAD` of your checkout. A build without a resolvable revision omits the commit rather than guessing.
- The **Config DB schema** line is the `user_version` of the resolved configuration database, read directly. It is omitted when no database is resolved or the database is unreadable.
- The **analyzer drift** line follows immediately, because it qualifies the commit printed above.
- The **version drift block** and the **CONFIG DRIFT** line are only present when something is wrong.

CONFIG DRIFT

The probe compares a fingerprint of what the server hands your harness - the composed instruction suffix, the effective tool-set spec, and the session auto-refresh mode - as captured at server start against a fresh read of the active profile. A change that does not alter what the harness receives (a profile name, or switching to a profile that composes an identical configuration) does not fire.

When it fires, the message is:

```
⚠ CONFIG DRIFT: the active MCP profile changed since this server started - restart/reconnect the aicb MCP server to load the current configuration (tools, instructions, session auto-refresh mode). In-session the skill-derived instructions never refresh, the auto-refresh mode is fixed at start, and a GUI edit does not update the tool list until restart.
```

The server also writes this message once to its error stream the first time it observes the drift. **What to do:** restart or reconnect the client. The protocol has no "instructions changed" notification, so the instructions a client received at the handshake cannot be replaced while the connection lives; the client's tool list is cached too, and no list-changed notification is sent.

VERSION DRIFT

This probe compares the binary against the configuration database and reports two independent findings:

- **Schema drift.** The database schema is newer than the highest migration this binary knows: a newer build migrated the database, and this binary lags it - rebuild or reinstall the standalone tool. The opposite case is reported as a pending migration: the database is older than this binary's migrations, and running the application, the GUI or a `--db-path` start migrates it.
- **Tool pool drift.** The active profile persists tool names this binary does not register. The message names up to ten of them (quoted, with `and N more` beyond that), because two causes look identical and their fixes are opposite:
- the profile was written by a **newer build** - rebuild or reinstall the standalone tool;
- the tool was **retired** and the profile still names it - re-save the profile in the GUI's `MCP Profiles` panel, which rewrites the selection without it. A rebuild would only repeat the message.

The stale entry is inert either way: an unknown name is ignored when the tool set is resolved.

Detection is on **tool-name level**. A new parameter added to an existing tool is not detected by this check - the build commit is the finer-grained signal for that. The version drift block is output-only, so it appears without reconnecting.

ANALYZER DRIFT

The third probe answers the axis the other two cannot see: this binary's build commit against the HEAD of the repository that holds the solution you asked it to analyze. A fact producer that got smarter since this build shows up here even though tool names and source files are unchanged. The probe reports every outcome, not only the bad one - the four measured ones below, and an unmeasured one described after the table:

Report	Meaning	What to do
`Analyzer: IN SYNC with the analyzed repo`	The binary was built from the current HEAD	Nothing - this is a proof of currency
`Analyzer: this binary is N commits AHEAD of the analyzed repo's HEAD`	Built from work the repository does not have yet; the analyzer is newer, not older	Nothing
`Analyzer: this binary was built from a commit N commits BEHIND ... but NOT ONE of the commits behind touches an analyzer project`	The binary is behind, but the analyzer's own code is unchanged across that distance	Nothing - the gap is no reason to rebuild

Report	Meaning	What to do
`⚠ ANALYZER DRIFT: ... N of them touch an analyzer project`	The analyzer's code moved since this build; facts it produces - fan-in, dead code, side effects - may be the older analyzer's, including a zero that the newer one fills	Rebuild or reinstall the standalone tool

If the distance cannot be characterized - for example the second measurement failed - the warning is kept without the qualifier, so a distance that cannot be judged stays as loud as before. The probe is silent when there is no analyzed session, when a repository does not know this build commit (any foreign solution), and whenever the binary carries no commit id. A git call that does not answer within two seconds is not silence: the line reads `Analyzer: drift UNKNOWN - git did not answer within 2 s in <directory>, ...` and asks you to call `server_info` again. If only the second measurement timed out, the warning stays and says that whether an analyzer project is touched is unknown. Public installer, ZIP and dotnet-tool release builds deliberately omit that id, so analyzer-drift output is unavailable there.

The measurement is a local, read-only `git rev-list --count` query in the solution's directory, with a two-second ceiling; a timeout is reported as `drift UNKNOWN`, any other error stays silent. It probes at most four distinct solution directories and stops at the first repository that can resolve the commit.

Besides these three drift signals, every session-bound answer carries a **staleness** note when source files changed after the analysis - that is a different signal with a different remedy; see "Sessions and staleness".

11.3 What the server records about tool calls

The server records one row per tool invocation into the local configuration database. The purpose is to improve the tool surface on measured usage rather than on assumptions; the recording is deliberately limited to shapes and contains no code.

The recorded row

Field	Content
Timestamp	UTC time of the call
Tool name	The tool that was called, `(unknown)` when it could not be determined
Session reference	**12 hexadecimal characters** from a SHA-256 hash of the session reference - never the raw path or id. A solution path is normalized first, so the slash and backslash spellings of one path give one reference
Success / error	Whether the call succeeded, and whether it failed
Error class	The exception **type** name
Result size	The summed length of the text blocks of the answer; empty when the answer carried no text
Duration	Latency in milliseconds
Client name / version	Taken from the protocol, as the client identified itself at connect time
Server version	The version of the server that served the call
Protocol version	The negotiated protocol revision
Alias applied	Set when the server accepted an alternative parameter name in place of the tool's real one
Facet	The resolved facet id the call addressed, or `(unresolved)` for a spelling the catalog does not know; empty when no facet was given
Error message	The failing exception's own message - the one free-text field (see below). A call that failed because it named an argument the tool does not have ends the message with `[unrecognized arguments: <names>]` (at most three names)

Field	Content
Child calls	For a `batch` or `measure` call: a per-tool tally of what it dispatched, for example `{"find_usages":{"n":3,"e":0}}`
Solution size	The number of projects and files of the solution the call was answered against; empty when the call resolved no session
MCP profile	The id of the MCP profile the server was serving when it admitted the call; empty for `aicb call`
Auto-refresh mode	The mode the server runs (`off` , `reactive` , `proactive`), after `AICB_MCP_AUTO_REFRESH` ; empty for `aicb call`
Unrecognized arguments	The names of arguments the call sent that the tool does not have (at most three), on success and failure; names only, never values
Client session	A random 12-hex id drawn once per server start, so one id is one MCP connection; it does not separate the subagents of one client session; empty for `aicb call`

What is **not** recorded: argument values, source code, file paths, and the content of any answer. Argument names are recorded only where the tool does not declare them. The session reference is only read in order to be hashed; a `facet` value is only read in order to be mapped to a fixed catalog id.

The row is written for **every** call on the `tools/call` pipeline, including calls the active profile refuses. A refused call is a curation signal, not noise, so it is recorded like any other.

Recording is **fail-open in both directions**: a telemetry defect can never break a tool call. On the error path the row is written first and the error is then re-thrown - the recording never swallows a tool error.

Where the data lives

The rows go into the `tool_calls` table of the configuration database - the same file the desktop application uses, at `%APPDATA%\AIContextBuilder\user-data\aicb.acb` unless `--db-path` says otherwise. The table carries an index on the tool name and one on the timestamp.

For ad-hoc queries, the columns are: `id` , `ts` , `tool_name` , `session_hash` , `ok` , `is_error` , `error_class` , `result_chars` , `duration_ms` , `client` , `server_ver` , `protocol_version` , `client_name` , `client_version` , `alias_applied` , `facet` , `error_message` , `child_calls` , `solution_projects` , `solution_files` , `mcp_profile` , `auto_refresh_mode` , `unrecognized_args` , `client_session` . The legacy `client` column is no longer written - the protocol identity replaces it.

A server **without** a usable database records nothing at all: it gets a no-op sink, no file is created, and `usage_report` answers `available: false` with a note.

Reading the record: `usage_report`

`usage_report` reports the server-side telemetry of every `tools/call` invocation - cross-client, not just this client's transcript.

Parameter	Default	Meaning
`topTools`	`30`	Cap on the per-tool breakdown, ranked by call count. Clamped to 1..200
`sinceDays`	not set	Only count calls from the last N days. Clamped to 1..3650; without it the whole log is read

The answer contains overall totals (total calls, error count and rate, distinct tools and sessions, client connections, first and last timestamp, the clients and server versions seen) and a per-tool breakdown ranked by call count:

Field	Meaning
`calls` , `errors` , `errorRatePct`	Call count, error count and error rate of the tool

Field	Meaning
`avgDurationMs`, `p50DurationMs`, `p90DurationMs`, `maxDurationMs`	Latency distribution. The percentiles are nearest-rank; they are the honest read for a right-skewed distribution that a single slow call drags upward
`avgResultChars`, `p50ResultChars`, `p90ResultChars`, `maxResultChars`	Response size distribution in characters
`errorClasses`	Histogram of the exception types behind the errors. A guided `McpException` is a failure the tool raised on purpose (an expired session, an unknown token); any other type is a defect suspicion
`argumentBindingErrors`	The subset of errors where the caller named an argument the tool does not have - the call never reached the tool. Neither a defect nor a refusal; a value here means the argument surface confused somebody
`ignoredArgumentCalls`	Calls that answered although they sent an argument the tool does not have, so that argument had no effect - the success-side twin of `argumentBindingErrors`; omitted when there were none
`clientSessions`	A total, not a per-tool field: how many MCP connections the recorded calls came from (`distinct` `calls` - the calls that carry a connection id - and a `note`). `distinctSessions` counts session references, and on a call that passes the solution path every chat on that solution shares one; one connection id does not separate the subagents of one client session. Omitted while no call in the window carries an id
`aliasApplied`	How often the tool was called with a parameter's accepted alias instead of its real name, i.e. how often the server silently corrected the caller
`poolCoverage`	How much of the active tool pool this traffic covers: `poolSize`, `poolToolsFired`, the named `poolToolsNeverCalled`, plus `outOfPoolCallsIncluded` and `outOfPoolTools`
`protocolVersions`	Share of calls per protocol revision, with `preInstrumentationCalls` for rows recorded before the instrumentation existed
`clientEras`	Cross-tab of protocol revision against client name and version, with `clientEraGroupsTotal` and `clientErasTruncated` when the list is capped
`facets`	How often each facet was addressed; omitted until a call has addressed one
`subQueries`	The per-tool tally of sub-queries dispatched through `batch` (see below); uncapped
`windowSinceIso`	The cutoff actually applied when `sinceDays` was given, so an empty window is never mistaken for an empty table
`recordedVia`	The scope of every figure in the answer

Read the scope before the numbers. It is repeated in the `recordedVia` field of the answer because a client caches the tool description until it reconnects:

- The recording sits on the `tools/call` **pipeline only**.
- A sub-query inside `batch` or `measure` does **not** get a row of its own; a recent-enough parent row carries the per-tool tally instead, reported under `subQueries`. Two boundaries of that block: a `measure` sub-query is left out (it runs the tool fully, but the caller consumes the size of the answer, not the answer), and a sub-query naming a tool the dispatch registry does not know is counted under the single literal `(unknown)` - a typo cannot grow the key space.
- A one-shot `aicb call` writes the same row under the client name `aicb-call` when it is given `--db-path`, and nothing without one.

So **every count is a lower bound**, and a tool reported at 0 was "not called through this door" rather than "not called". Read pool coverage from `poolCoverage`, not from `distinctTools`: the latter counts every tool that was called, including ones outside the current pool, so holding it against the pool size overstates coverage. A name in `poolToolsNeverCalled` carries its own note:

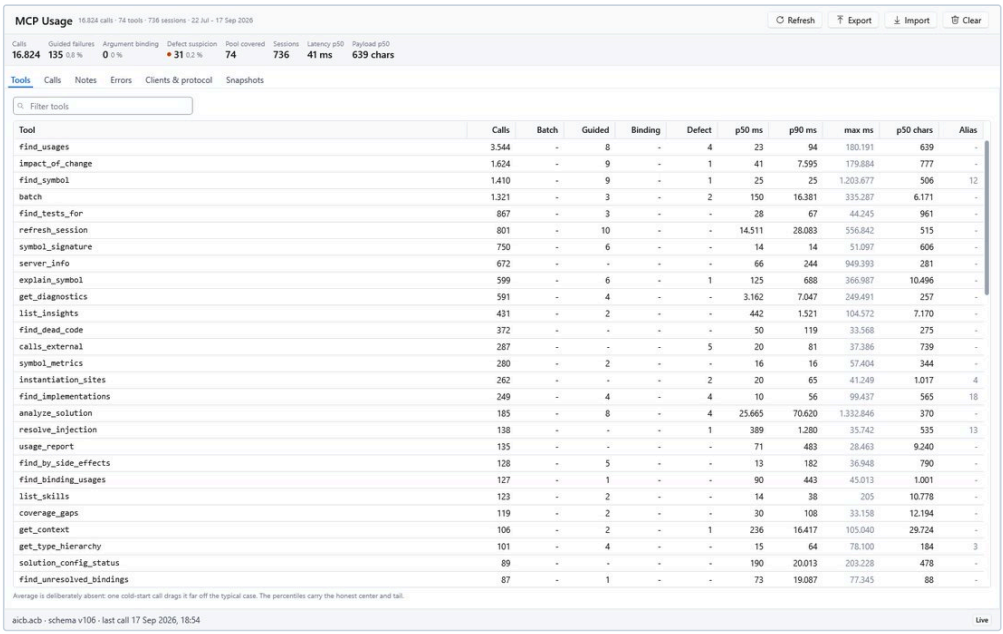
it was never called *top-level*, and the list does not subtract the `subQueries` - a tool can appear in both. The note also names the span the list was measured over - the window cut, the oldest recorded call in it and how long ago that was; an empty window says the span is unknown, so the list is unmeasured, not unused.

`usage_report` is **not available inside** `batch` **or** `measure`. It is a read-only tool, but it is deliberately not part of the dispatch registry those two serve.

When there is nothing to report, the answer says which case applies: `available: false` with a note when the server is running without a database or the table is not readable; `available: true` with "No tool calls recorded yet." when the table is empty.

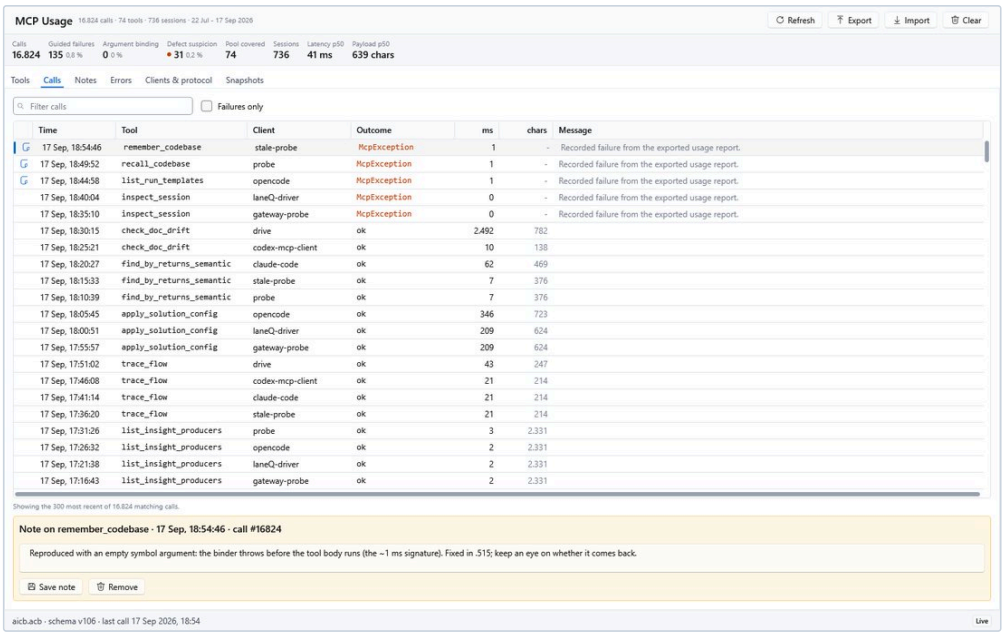
The MCP Usage panel in the desktop application

The desktop application shows the same data under `MCP Usage`. The `Tools` tab is the aggregate: one row per recorded tool with its call count, the `Batch` sub-query count, guided refusals and suspected defects kept apart, the p50/p90 latency and response size, and an `Alias` count.



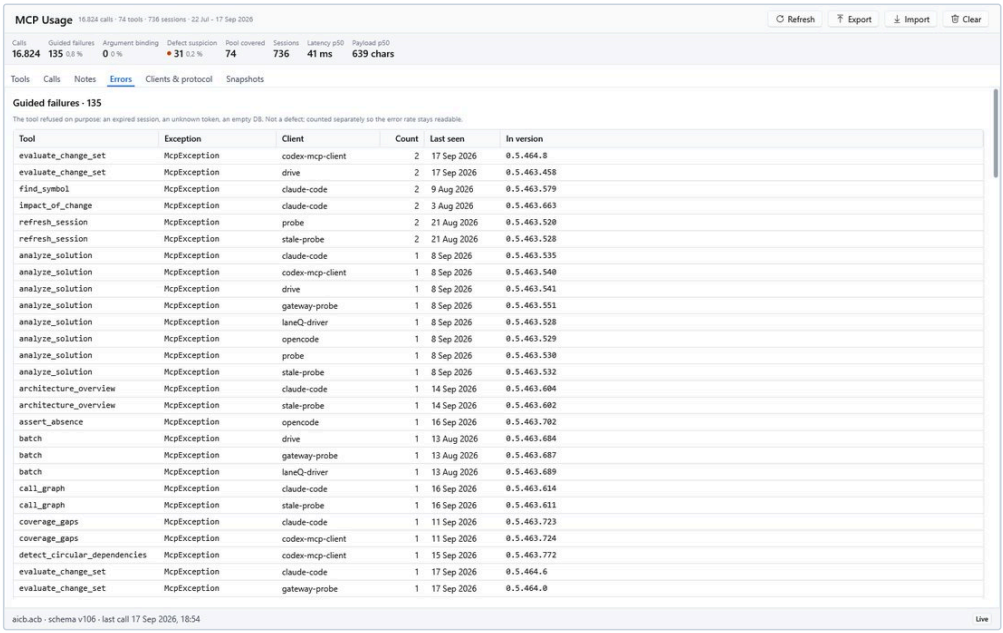
MCP Usage: every recorded tool with its call count, guided refusals separated from suspected defects, and p50/p90 latency and response size

The `Calls` tab is the raw log under the aggregates: one row per recorded call with time, tool, client, outcome, duration, response size and the captured message. It is also the only region whose rows carry an id, which is what the `Notes` tab refers to - a note you write on a call points the log at it.

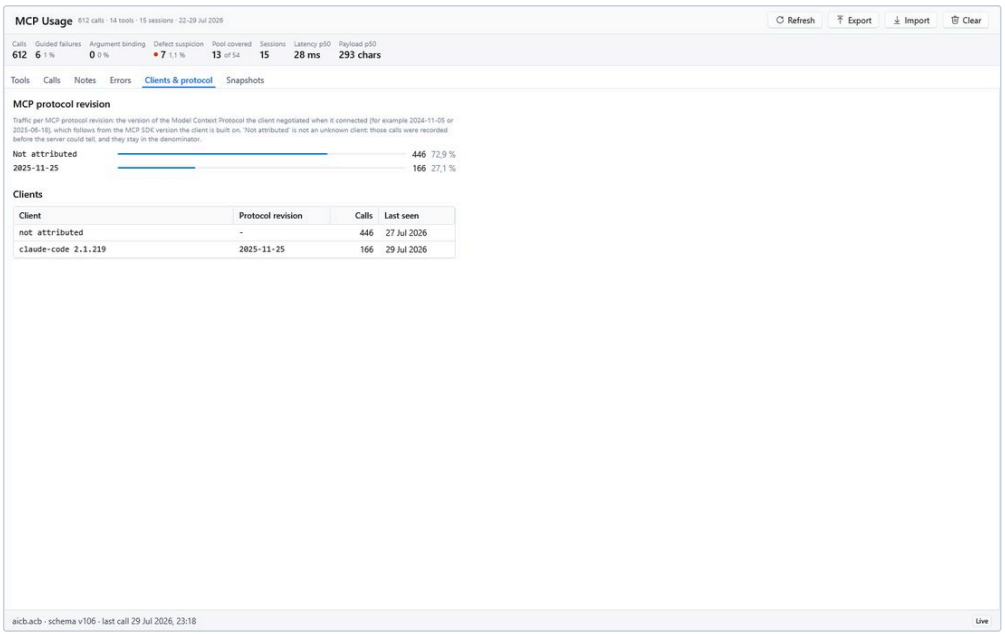


The Calls tab: each recorded call with time, tool, client, outcome, duration, response size and the captured message

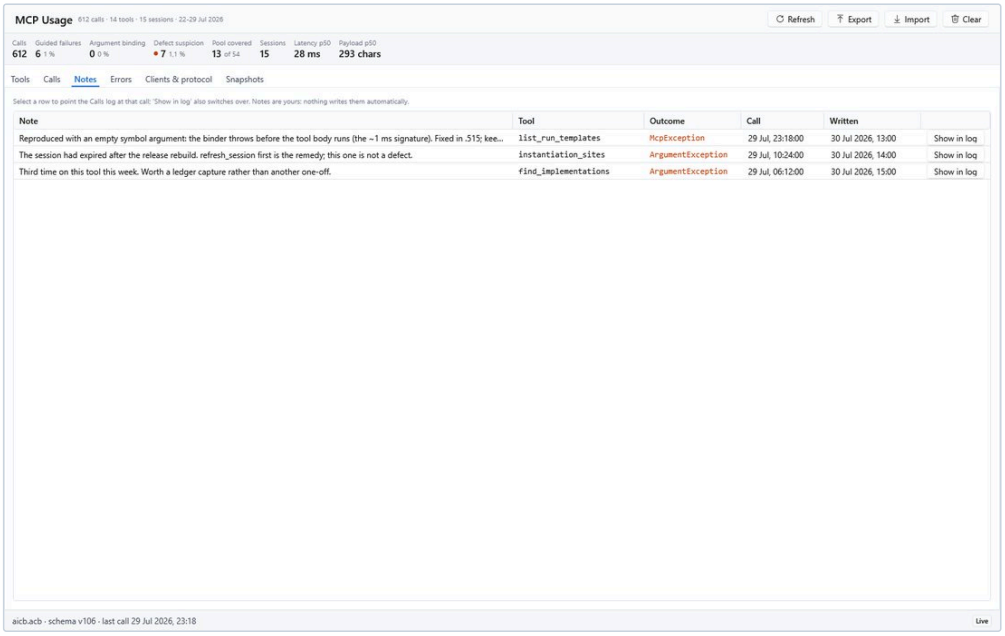
The `Errors` tab separates guided refusals from suspected defects. It has three groups: **Guided**, **Defect**, and **Binding**. Binding failures are a separately displayed subset counted out of Defect: the caller named an argument the tool does not have, so the call never reached it. Each group shows tool, exception type, client, count, when it was last seen, and the server version it was last seen in.



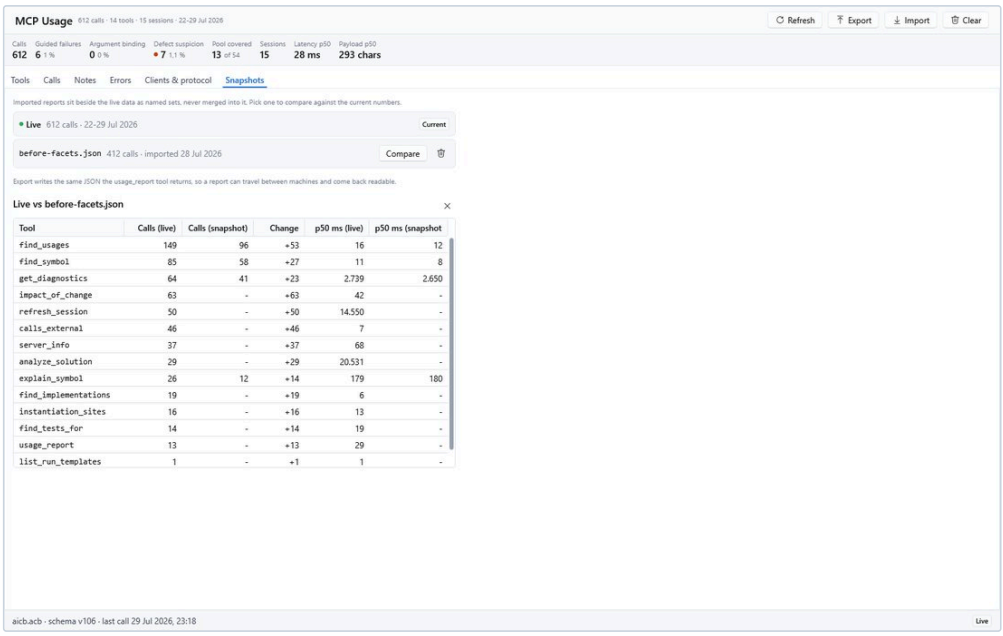
The Errors tab: intentional refusals separated from suspected defects, per tool, exception type and client



The Clients & protocol tab: which clients and protocol revisions have connected



The Notes tab: free-text notes attached to individual calls



The Snapshots tab: an imported usage report beside the live numbers

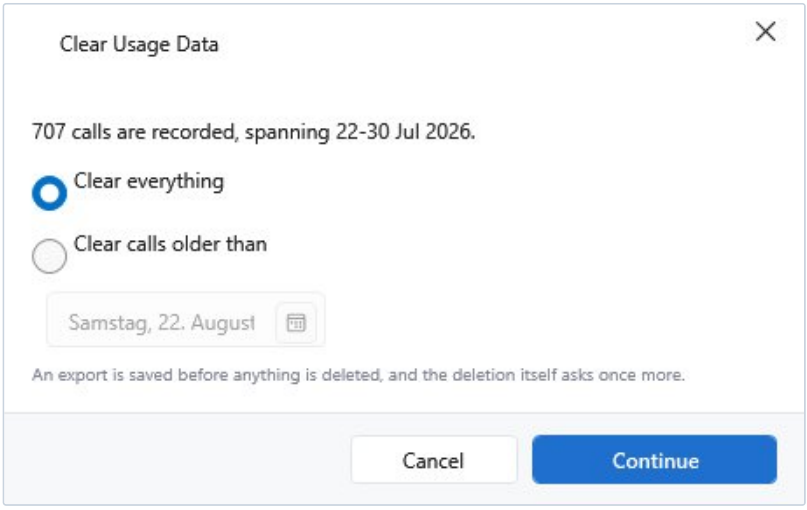
`Export` writes the current report as JSON. The file carries exactly what the `usage_report` tool returns - the same serializer produces both, only the file form is indented because a person opens it - so a report can travel between machines and be imported again. `Import` adds an exported report as a named snapshot beside the live data; imported sets are never merged into the live numbers. `Refresh` (or `F5`) reloads from the database.

Deleting recorded data

Deleting telemetry is a desktop-application action, deliberately: the MCP server is read-only by contract and exposes no delete verb, and there is no CLI verb for it either. In a headless setup the only way to remove the data is to replace the database file.

In the application, `Clear` on the `MCP Usage` panel runs this sequence:

1. A scope dialog asks what should be deleted - everything, or calls older than a date.
2. An **export is forced**: you choose a file, and if you cancel the file dialog the whole deletion is cancelled. Nothing is deleted before a report has been written.
3. A destructive confirmation names how many calls will be deleted, the export file name, and the notes that are lost with them.
4. The calls are deleted and the panel reloads.



The scope dialog of Clear Usage Data

Notes die with their calls: a note is attached to one recorded call, and deleting calls deletes their notes. The export contains the aggregate and therefore **not** the notes - the confirmation says so when any are affected.

Privacy and what leaves your machine

The recording is built to be safe even if the database is shared or public:

Measure	Effect
The session reference is hashed	SHA-256 over the UTF-8 value, first six bytes - 12 hexadecimal characters. The raw `.sln` path or session id never reaches the database
The facet is normalized	Stored as a resolved catalog id or the one literal `(unresolved)`, never as the caller's free text
The sub-query keys are bounded	Only names the dispatch registry knows; everything else falls back to `(unknown)`
The error message is the only free-text field	Capped at **500 characters including the truncation marker** `...`, enforced twice - when the row is shaped and again at the write boundary. A whitespace-only message becomes empty, which is deliberately different from an empty string
Only the outermost exception message	Never an inner one; an aggregated chain would blow past any cap and bury the actionable sentence
`AICB_MCP_ERROR_TEXT=off`	Records only the exception **type** - the pre-message contract, intended for a database that is shared rather than local. Case-insensitive, and read per call, so it can be flipped without restarting the server
No sink without a database	A server without a usable configuration database writes nothing

The child-call tally is not capped; its bound is the 16 sub-queries a `batch` call may carry.

There is no environment variable and no setting that turns the recording off. The only state in which nothing is recorded is a server running without a usable configuration database - which also costs you the profile-aware behavior, the stored per-solution configuration and the report itself. `AICB_MCP_TELEMETRY` is not read by the MCP server and has no effect on this recording.

What leaves your machine: **nothing**. The server registers the stdio transport only - there is no HTTP or SSE transport - and it references no analytics, crash-reporting or HTTP package. Its four outputs are the stdio JSON-RPC channel back to the client that made the call, diagnostics on the error stream, local SQLite writes, and a local read-only `git` query for the analyzer drift probe (see above). There is no telemetry upload, no analytics endpoint, and no configuration that could enable one: the capability is not in the assembly.

Note: this describes the aicb server itself, as shipped. The Model Context Protocol package it uses is a third-party dependency and is not audited here.

11.4 Installing the symbol guard into your agent harness: `install_agent_hooks`

`install_agent_hooks` is the one MCP tool that writes into the configuration of the **calling harness**. It installs the symbol guard, which refuses a C# symbol question aimed at a text search and redirects the caller to the tool that answers it properly (with the two exits described under "Two ways past the refusal").

It is part of the navigation core in every profile, and that is deliberate: the session hint that offers this tool reaches every client, so a tool the default profile could not call would be an offer that fails when accepted.

Because it writes, it is **not** reachable through `batch` or `measure`: a call you believe is a read can never perform an installation.

Parameter	Required	Meaning
`sessionId`	yes	A session id, or the absolute path of a `.sln`, `.slnx` or `.slnf` file (self-init)

Parameter	Required	Meaning
`harness`	no	`claude-code`, `codex` or `opencode`. Omit it to install for the harness the calling MCP client belongs to
`force`	no, default `false`	Overwrite an existing guard script and rewrite an existing wiring entry

What is installed per harness:

Harness	Files written	Wiring file	Merge
`claude-code`	`.claude/hooks/aicb-symbol-guard.mjs`	`.claude/settings.json`	Hook list entry with matcher `Grep|Bash|Edit|Write|MultiEdit|Read`; the command uses `$CLAUDE_PROJECT_DIR`
`codex`	`.codex/hooks/aicb-symbol-guard.mjs`	`.codex/hooks.json`	Hook list entry with matcher `Bash|apply_patch`; the command uses an **absolute** path (Codex has no project-directory variable, so it becomes invalid if the checkout moves)
`opencode`	`.opencode/plugins/aicb-symbol-guard.ts`, `.opencode/plugins/guard-wiring.mjs`, `.opencode/plugins/aicb-symbol-guard.mjs`	`opencode.json` in the project root	Plugin list entry

Behavior worth knowing:

- **The harness is resolved before the session.** A bad `harness` name is answered without loading a solution. If you omit `harness` and the client announced no name, or a name aicb has no wiring for, the tool **refuses** rather than guessing - a default would write a working-looking configuration into a harness that never reads it.
- **It is idempotent by default.** An existing guard script or wiring entry is kept (`Unchanged`, "already there - left untouched") unless `force=true`. Your edited exemption list is the obvious thing to lose here, which is why the default is conservative.
- **It writes only inside the directory of the session's solution.** There is no path parameter, and the segments come from fixed lists. Because a `sessionId` may itself be a solution path, the reachable set is "any directory holding a loadable solution" - not only the project the caller was already working in.
- **It never writes** `.mcp.json` (that is the file the server is currently being served from) and never writes the agent skills (that is `aicb init`'s decision).
- **Order and atomicity.** All hook scripts are written first, then all wiring entries, so a configuration can never point at a file that did not appear. Files are written atomically (temporary file, then move). An I/O error on one artifact degrades to a `Refused` line instead of aborting the whole run.
- **Consent is structural.** The tool asks no question: you (or your agent) must call it explicitly, an existing guard or wiring is preserved without `force=true`, and the write surface is fixed.
- **Nothing takes effect until the client reconnects.** The guard runs in the client, which read its configuration at startup. The answer says so in its `nextStep` field.

The answer is a JSON object with `harness`, `harnessName`, `projectDirectory`, `succeeded` (false when any artifact was refused), `artefacts` (one entry per file with `path`, `outcome` and `detail`), and `nextStep`. The outcomes are `Created`, `Updated`, `Unchanged` and `Refused`. After a successful install, restart or reconnect the client, then call `refresh_session` so the session stops reporting the guard as missing.

If you prefer to install the guard from the command line, `aicb init` writes the same artifacts; `aicb init --hooks none` installs no enforcement, and `--hooks claude-code|codex|opencode|all` installs it deliberately.

11.5 MCP resources

Besides the tools, the server registers **resources**: content a client fetches with `resources/list`, `resources/templates/list` and `resources/read` instead of a tool call. There are ten, in three groups.

URI	Name	MIME type	Parameter	What it returns
`acb://remembered`	`remembered_codebases`	`application/json`	-	The remembered codebases with their latest-snapshot timestamp, file-set hash and whether a model payload is present. An empty array without a default config DB
`acb://templates`	`run_templates`	`application/json`	-	The run templates available in the default config DB (built-ins plus custom ones) with id, name and flags. Built-ins only without a default DB
`acb://snapshots/{hash}`	`snapshot_by_hash`	`application/json`	`hash`	Metadata of the remembered-codebase snapshot with that file-set hash - id, solution id and path, name, type, creation time, file-set hash, payload version, whether a model payload is present, and the debt figures. **Metadata only, no payload blobs.** Errors on an empty hash, without a default DB, and when nothing matches
`acb://schema/constellation-v1.json`	`constellation_schema`	`application/json`	-	The embedded JSON schema (Draft 2020-12) for constellation files. Static, independent of any database
`acb://docs`	`aicb_manual`	`text/markdown`	-	The operating manual's directory: every page with a one-line summary, a token estimate and the reading routes
`acb://docs/{topic}`	`aicb_manual_page`	`text/markdown`	`topic`	One manual page or a resolved route. Page ids: `overview`, `install`, `first-context`, `navigation`, `solution-config`, `glossary`; route: `init`
`acb://sessions/{id}/markdown`	`session_markdown`	`text/markdown`	`id`	The rendered context document of an analyzed session

URI	Name	MIME type	Parameter	What it returns
`acb://sessions/{id}/insights`	`session_insights`	`application/json`	`id`	The insights of an analyzed session - the same content `list_insights` returns
`acb://sessions/{id}/architecture`	`living_architecture`	`text/markdown`	`id`	The structural-only overview of the whole solution, with the format preamble and without per-type source; test projects excluded; its layer map is bounded as in `architecture_overview`
`acb://quality-profiles`	`quality_profiles`	`application/json`	-	The built-in insights profiles (default, strict, disabled) with their thresholds and producer toggles

Three properties to know when you build a client or an integration:

- **Resources are not tool-curated.** The active MCP profile narrows `tools/list` and `tools/call`; it does not narrow `resources/*`. Every resource is served regardless of the active pool.
- **The URIs live in two lists.** The five resources with a fixed URI - `acb://remembered`, `acb://templates`, `acb://schema/constellation-v1.json`, `acb://docs`, `acb://quality-profiles` - appear in `resources/list`. The five templated ones - `acb://snapshots/{hash}`, `acb://docs/{topic}` and the three `acb://sessions/{id}/...` - are readable but do **not** appear in `resources/list`; they are exposed through `resources/templates/list`. All ten are therefore discoverable, but a client that only asks `resources/list` sees five of them. That is why `acb://docs` exists as a static entry beside its own URI template: it is what makes the manual reachable for a client that has not seen the template.
- **The database-bound resources read the default configuration database** (the same one the DB-bound tools default to) and build a short-lived provider per call. Without a default DB they return an empty array or the built-ins; `acb://snapshots/{hash}` errors instead of guessing. The session-bound resources need a live session id.

A deliberate difference from the tools: the resources carry no tool-reachability note. A resource is fetched once and cached by the client, so content that varies with the live tool pool would freeze a claim that becomes wrong after the next profile change.

12 Troubleshooting the MCP server

This chapter is for the person who runs `aicb` as an MCP server and reads what an agent reports back. The server has no window of its own: nearly every problem surfaces either as a message in a tool answer, as a client-side error such as `MCP error -32000: Connection closed`, or as a line in the stderr log of your MCP client. The sections below are ordered by symptom. Each one names the message you will see, the cause, and what to do about it.

Two rules make the rest of this chapter easier to use:

- **Do not judge freshness from the tool you called.** Every session-bound answer that may be out of date carries its own note (a `staleness` JSON member or a `<!-- staleness: ... -->` comment). Read the note, not the clock.
- **Read the leading note before the list.** Several tools put a caveat about the analysis run, or about an empty scope, ahead of their data. That caveat is often the actual answer to "why does this look wrong?".

12.1 The client does not list any `aicb` tools

Symptom. Your MCP client shows no `aicb` server or no `aicb` tools, or it reports that it cannot start the command.

Cause and solution. The server is started by the client from an entry in the project's `.mcp.json`:

```
{
  "mcpServers": {
    "aicb": {
      "command": "aicb",
      "args": ["mcp"]
    }
  }
}
```

`"command": "aicb"` requires `aicb` to be on your `PATH`. The installer puts it there (unless you untick that task); the portable ZIP does not, because it only unpacks files. For a portable copy, write the full path into the JSON and double every backslash:

```
{ "mcpServers": { "aicb": { "command": "C:\\path\\to\\this\\folder\\cli\\aicb.exe", "args": ["mcp"] } } }
```

Check the setup step by step:

1. Open a **new** terminal and run `aicb --version`. If the command is not found, use the full path to `aicb.exe` in the `.mcp.json` entry instead.
2. In your project directory, run `aicb init`. It writes the `.mcp.json` entry and the agent skill, and reports per file what it did. It never overwrites an existing artifact (`aicb init --force` does).
3. Restart or reconnect your MCP client. A client reads its server list at startup; a running client will not see a new entry. Consoles, editors and agents that were already open also see a changed `PATH` only after a restart.
4. Ask the agent to call `server_info`. If that answers, the connection works.

Note. OpenCode does not read the shared `.mcp.json`; it discovers MCP servers only through its own `opencode.json`. `aicb init` writes the `aicb` entry there when the project uses OpenCode or `--hooks opencode` names it; by hand, the entry is `"mcp": { "aicb": { "type": "local", "command": ["aicb", "mcp"], "enabled": true } }`.

Two installations on one machine. If `aicb` exists twice (for example the desktop installer and a .NET global tool), only the **first** one on `PATH` is ever started, and updating the other one changes nothing a client runs. `aicb init` warns when it finds more than one and lists which is used and which is ignored. Keep one: uninstall the global tool, or uninstall the desktop app under Windows Settings > Apps.

12.2 Protocol revision and transport

The server speaks both current MCP protocol revisions over stdio from the same tool pool. The client picks the one it knows:

Revision	Entry point
`2026-07-28`	`server/discover` - stateless, every request carries its own metadata
`2025-11-25`	`initialize` handshake

This matters because MCP has no fall-forward: a client that speaks only an older revision cannot reach a server that speaks only a newer one. `aicb` answers both entry points, and it also serves older revision handshakes (for example `2025-06-18`, which some clients still negotiate), so you do not have to configure or match a revision.

Two limits are worth knowing:

- **stdio only.** There is no HTTP/SSE transport, so the HTTP-specific parts of the newer revision (session headers, resumability, OAuth) do not apply here.
- **Server instructions are delivered, not pushed.** After you change the active profile in the GUI, restart the server. A running client is not told that the instructions changed.

Note. The tool list has a lifetime of 15 minutes and the server never sends a `tools/list_changed` notification. A profile change made outside the running server (for example in the GUI) reaches it only after a restart; until then `server_info` reports it as config drift. After the restart, a client picks up the new list when its cached copy expires or when it reconnects.

12.3 `Tool '<name>' is not in the active profile`

Symptom.

```
Tool '<name>' is not in the active profile - call list_mcp_profiles to see the profiles and their tools, then
restart the server with 'aicb mcp --mcp-profile <id>' to run under a different one.
```

Cause. The exposed tool set is decided by an MCP profile. The Default profile serves 54 of the 82 registered tools, while `Full Select` serves the 72-tool lean core. The remaining routes are explicit: `compare_public_api` needs the `api-surface` bundle (or an override), and nine uncatalogued session/memory/database tools are reachable only through `AICB_MCP_TOOLS`.

Solution.

1. Call `list_mcp_profiles` if the active profile exposes it. It lists every profile with its id, its tool selection and whether it is active. A custom profile can switch this core tool off; `server_info` is the only core tool that cannot be disabled.
2. Restart the server with the profile you want, for example `aicb mcp --mcp-profile mcp-profile/full`. The pin is process-local: it does not change the profile stored in the config DB. The same ids are visible in the GUI under `MCP Profiles`.
3. There is **no runtime switch**. The active profile is fixed when the server starts, because the tool list must not depend on a previous tool call. Changing it always means a restart.

`AICB_MCP_TOOLS` **- the operator override.** The environment variable `AICB_MCP_TOOLS` takes precedence over the active profile. `all` exposes every registered tool, literal `lean` exposes the 72-tool lean core, and a comma-separated list narrows the set to those tool classes. Empty or unset means no environment override, so the config file and active profile decide. A list prefixed with `methods:` names individual tools instead of classes.

`list_mcp_profiles` **needs a config DB.** On a server that started without one it answers:

```
MCP profiles require a config DB. Start the server with: aicb mcp --db-path <db>.
```

Inside `batch` **and** `measure`**.** Sub-queries follow the same active pool as direct calls. A sub-query whose tool is read-only but outside the profile is refused individually with a message that names the same remedy:

```
'<tool>' IS read-only and dispatchable, but it is NOT in the active profile: <batch|measure> sub-queries follow the same active pool as tools/call, so a narrowed profile is narrowed here too (a direct call is refused by the same rule). Run list_mcp_profiles to see the profiles and their tools, then restart the server with 'aicb mcp --mcp-profile <id>' - or widen AICB_MCP_TOOLS - to expose it.
```

Note. A blocked call attempt is still recorded in the usage telemetry, so `usage_report` can show that an agent reached for a tool the profile did not expose.

12.4 The agent sent an argument name the tool does not have

This is the most dangerous MCP error class, because one of its two forms produces no error at all. An MCP client binds the arguments it recognizes and drops the rest without a word.

Form A - the wrong name belongs to a required parameter. The call fails, and the server makes the failure diagnosable:

```
<ExceptionType>: <cause> - This is an ARGUMENT-BINDING failure: the call never reached the tool, so retrying it unchanged fails identically. <tool> takes: <parameters, required marked>. You sent: <...>. Not a parameter of this tool: <...>.
```

If the parameter inventory cannot be read, the middle part falls back to:

```
Compare your arguments against this tool's input schema in tools/list - argument names are NOT uniform across the tool surface.
```

Form B - the wrong name belongs to an optional parameter. The call succeeds and answers a **different question** than the one asked. The server therefore appends a disclosure to every successful answer that contains an argument it does not declare:

```
<!-- ignored-arguments: <names> are not parameters of <tool> and had NO effect on this answer - it was computed as if they had not been sent. <tool> takes: <parameters>. -->
```

Read that comment literally: the answer is correct **for the call as it was bound**, not for the call as it was meant.

Example. A query that used `namespaceFilter` on `find_by_concurrency_risk` returned the solution-wide answer and reported `scope: "solution"`, including matches from the namespaces the caller believed were excluded. The parameter is called `scope`. With the correct name the same query is restricted to the intended namespace.

What to do. Compare the parameter names you (or your agent) sent with the ones the tool publishes. The exact inventory is in the failure message, in the `ignored-arguments` comment, or in `tools/list`. The server also accepts `symbol` as an alias for several symbol-shaped parameters, including `interfaceName`, `typeName`, `query`, `interface`, `methodName` and `method`, and rewrites it silently; `usage_report` counts those rewrites as `aliasApplied`.

Limits of the disclosure. Up to 8 dropped names are listed, each reduced to identifier characters and capped at 64 characters. The disclosure stays silent for a tool that is unknown or publishes no arguments - a half-known inventory would accuse the caller on the strength of a failed lookup. The failure message caps the echoed cause and the parameter inventory at 400 characters each.

12.5 `MCP error -32000: Connection closed`

Symptom. A tool call answers `MCP error -32000: Connection closed`, and the server is unreachable for the rest of the MCP session.

What it is not. The obvious diagnosis - "a tool exception ended the stdio loop" - is wrong. It was tested against a real server process with seven failure shapes (an unknown session, a deliberate `McpException`, an unbindable argument type, an unknown argument name, and more): every one came back as a clean error response, and the server answered a liveness probe afterwards.

What it is. The **channel**. stdout is the JSON-RPC transport, and a single stray write to stdout - from the product, from a dependency, from MSBuild or Roslyn during an analysis - puts a non-JSON line into the protocol stream. A strict client treats that as a protocol violation and tears the connection down. The process itself stays alive and healthy, which is why looking for a crash finds nothing.

What the product does. Two guards protect the channel: `Console.Out` is redirected to stderr, so no code path can put a byte into the JSON-RPC stream, and exceptions that no filter can see (thread-pool callbacks, timers, background tasks) are reported on stderr before the process ends:

```
[aicb] FATAL: unhandled exception - the aicb MCP server is terminating: <exception>
[aicb] WARNING: unobserved task exception in the aicb MCP server: <exception>
```

What to do. Read the **stderr log of your MCP client**. Where that log lives is decided by the client, not by `aicb`. The stray line or the exception is in it. Then restart the server and report what you found.

12.6 Answers describe the code as it was before your edit (staleness)

Symptom. The agent reports something that does not match the code you just wrote: a type is `not_found` although it is in the editor, or a fan-in count does not know a call site you just inserted. The answer carries a note - either a JSON member named `staleness` or a Markdown comment `<!-- staleness: ... -->` at the top of the answer.

The four texts. Read the one you got; they differ in cause and remedy:

Situation	Message (core)
The automatic re-analysis ran and succeeded	`source files had changed, so <this session> was re-analyzed automatically (incremental|full reload) before this call was answered - this answer reflects the code as it is now`
The automatic re-analysis failed	`source files changed since <this session> was analyzed and the automatic re-analysis FAILED (<reason>), so this answer comes from the pre-edit graph - call refresh_session to retry it explicitly`
No automatic re-analysis, live session	`source files changed since <this session> was analyzed, so this answer comes from the pre-edit graph - call refresh_session for a current one`
No automatic re-analysis, recalled session	`source files changed since <this session> was remembered, so this answer describes the code as it was - call refresh_remembered for a live re-analysis`

A `(last checked <timestamp>)` suffix names when the check last looked. The JSON member additionally carries `stale`, and, when an automatic refresh acted, `autoRefreshed` with `mode` (`incremental` or `full-reload`) or `autoRefreshFailed` with the failure type. A successful refresh means the answer is current - the note tells you what happened, not that something is wrong.

The auto-refresh mode. There are three modes, and they answer "who pays for the re-analysis":

Mode	Behavior
`off`	Drift is disclosed and left alone.
`reactive`	The server re-analyzes **before** answering, when source files have moved. This is the default.
`proactive`	A background watcher starts the refresh when a `.cs` file is saved, so the graph is usually current by the time a tool asks.

The mode is set when the server starts and cannot be changed while it runs. It comes from the active MCP profile (set it in the GUI) and can be overridden for one process with `AICB_MCP_AUTO_REFRESH`: `off`, `0`, `false`, `no` for `off`; `reactive`; `proactive`, `1`, `on`, `true`, `yes` for `proactive`. An unrecognized value is ignored, so a typo never switches the mode. Note: the legacy value `true` means `proactive`.

What to do.

- Under `reactive` (the default), a successful automatic refresh needs no action.
- After a failed automatic refresh, call `refresh_session` explicitly.
- After a **restore or build**, call `refresh_session` with `force: true`. The file-set check reads source files and cannot see build output, so an ordinary call may legitimately answer `changed: false`. `force` costs a full reload. An ordinary call already reloads when a restore rewrote a project's `project.assets.json` after the session was loaded.
- A recalled session (created with `recall_codebase`) has no live workspace, so `refresh_session` rejects it. Use `refresh_remembered` or `analyze_solution`.

Two different staleness axes. Distinguish them, because the remedies differ:

- **Source staleness** - `not_found` on a type you just wrote. The graph is behind your files. `refresh_session` fixes it; under `reactive` the server already did it.
- **Reference incompleteness** - a plausible but **too short** fan-in list, disclosed through `incompleteProjects`. The graph was built from source that could not resolve its references. Only this axis needs a build first, then `refresh_session(force: true)`.

`refresh_session` returns `{ changed, reason, session, mode }`. `mode` names the path taken: `incremental` means document texts were replayed into the warm snapshot without an MSBuild reload, otherwise `full-reload`. It is absent when nothing was re-analyzed.

12.7 A result looks wrong: the analysis run was incomplete

Symptom. A tool answer leads with:

```
THIS ANALYSIS RUN WAS INCOMPLETE: <n> of <m> scanned projects could not resolve their references (<up to 5 names>[, and <k> more]). <family-specific consequence> Restore/build the solution, then refresh_session and re-run this query.
```

Cause. Some projects could not resolve their references, usually because the solution has not been restored or built. Unresolved code still parses, so facts were produced - but the semantic edges into and out of it are missing.

Why it matters. The alarm rides on healthy-looking, non-empty lists as well, and that is the point: a plausible answer from an incomplete run is more dangerous than an empty one. Each query family gets its own consequence sentence, because the loss is different on each axis:

Query family	Tools	What the caveat tells you
Fan-in	`find_usages`, `impact_of_change`, `find_dead_code`, `find_production_dead`, `instantiation_sites`, `lifecycle_of`	The numbers are short by an unknown number of **ordinary** references, not merely exotic ones.
Effects	`find_by_side_effects`	An effect is derived by propagating along call edges, so methods read as **pure** that are not. Treat the result as a floor, never as a passed purity or clean-architecture check.
External calls	`calls_external`	A call whose target did not resolve records no external key. An empty result is no evidence that the API goes unused.

Query family	Tools	What the caveat tells you
Code traits	`find_by_code_traits`	`reflection` and `linq-in-loop` exist only where a target resolved; `throws` is pure syntax and unaffected. Treat a zero on the resolved traits as unmeasured.
Declarations	`find_implementations`, `get_type_hierarchy`	Interface and base-type identities may not bind, so declaration relationships can be missing or attached to the wrong name.
Resource leaks	`find_by_resource_leak`	Both the leak list **and** the `disposableCreationsChecked` denominator are short. A zero here is unmeasured, not leak-free.
XAML bindings	`find_unresolved_bindings`	The only axis whose loss runs in **both** directions: a count that is short, or bindings flagged that are fine. Never a clean bill and never a defect list.
Event subscriptions	`find_by_event_subscription`	Matches and the `availableEvents` suggestion list are both short, so the discriminator itself is unreliable.

At most 5 project names are spelled out; the rest is summarized as `and <k> more`.

A different zero: nothing was scanned. This is not a run alarm - the run may be fine, the scope was empty:

```
Nothing was scanned - the scope matched no unit: a mistyped or too-deep namespace prefix, or a scope holding only test projects while includeTests is false. This zero is not a finding about any code.
```

What to do. Restore and build the solution, then call `refresh_session` with `force: true` and re-run the query. `get_diagnostics` shows which projects are incomplete and how many diagnostics they took with them.

12.8 "Unknown session" and other session-resolution failures

A failed session resolution names its actual cause. The three shapes are mutually exclusive, and only one of them is an expired session:

What you passed	Message
A path with a solution extension, but no file there	`No solution file at '<x>'. The sessionId has a solution extension, so it was read as a path to self-initialize from - but no file exists there. Check the path (it must be ABSOLUTE), or pass a session_id returned by analyze_solution. This is NOT an expired session.`
An existing solution file, but this host cannot self-initialize	`'<x>' exists, but this host cannot self-initialize a session from a solution path. Call analyze_solution and pass the session_id it returns.`
Anything else	`'<x>' is neither a known session_id nor an absolute .sln/.slnx/.slnf path. If you meant a session_id: it is unknown or expired - call analyze_solution first. If you meant a solution: pass its ABSOLUTE .sln/.slnx/.slnf path to self-initialize.`

Two further messages tell you the session exists but has no live workspace:

- `Session '<x>' is recalled from memory (no live Roslyn workspace). Call refresh_remembered (or analyze_solution) for a live session.`
- `Session '<x>' is no longer live (its workspace was released). Call analyze_solution again.`

Cause of the second message. MCP sessions expire and are evicted. Defaults: a session lives 90 minutes from its last use, at most 8 sessions are held at once (the least recently used is evicted), and a background sweep frees expired sessions every 5 minutes. You can override the three values for one server process with the environment variable `AICB_MCP_SESSION`, a comma-

separated `key=value` list with the keys `ttlMinutes`, `maxSessions` and `sweepMinutes`, for example `maxSessions=4,ttlMinutes=120`. Keys that are not set keep their default; invalid or non-positive values are ignored.

12.9 Two analyses of the same solution at once

Symptom. A call hangs for a long time and then answers:

```
Gave up waiting for the solution registry after <N>s: the registry is <loading|reloading|applying edits to>
'<path>', started <M>s ago. This call never started - it was queued behind that operation, so retrying now
would queue again. Wait for the running load to finish, or analyze a smaller solution/filter (.slnf).
```

When a `.sln` path is passed to a tool directly (self-initialization), the equivalent message reads `Gave up waiting for the solution analysis after ...`.

Cause. There is one gate per solution path. A second caller waits instead of starting a second analysis. The message exists because a caller sees one pending answer and cannot tell a running load from a hung server.

What to do. Wait - do not retry, that only queues again. For very large solutions, analyze a `.slnf` filter file instead.

The wait is adjustable. The bound is 45 seconds by default and can be changed with the environment variable `AICB_MCP_LOAD_WAIT_MS` (milliseconds). A zero or negative value is refused and the built-in default applies. The GUI and the CLI wait without a bound, which is intended there; only the MCP server uses this limit.

12.10 Database-path errors

Tools that need a database validate their `dbPath` argument and answer with one of two messages:

```
dbPath is required (<purpose>).
dbPath is a directory, expected a file: <dbPath>
```

The `purpose` clause names which database is missing, for example `the memory DB to persist into / recall from`, `the aicb DB to save into / compare against` or `the aicb config/master DB to read/write the per-solution config`. The file **need not exist** - it is created and migrated on first use. Only an empty path and a path that names a directory are rejected.

12.11 Self-diagnosis

`server_info` - reachability, build commit and three drift signals

`server_info` answers in one call what the version number alone cannot. Its first line always carries the server name and version; non-release/development builds can additionally carry the **build commit**:

```
aicb MCP server (AIContextBuilder) v<version> (commit <sha>) - Roslyn-based .NET context generator. <license
notice>
```

When present, the build commit answers "is the binary I am talking to built from the code I just landed?". Compare it with `git rev-parse HEAD`; the reported SHA is the full one, so a short hash is a prefix of it. Public installer, ZIP and dotnet-tool release builds deliberately omit the commit from their informational version; development builds may include it. A build without a resolvable revision omits the field rather than guessing.

If a config DB is in use, the answer continues with its schema version:

```
Config DB schema: user_version=<n> (<path>).
```

Then come up to three drift signals. They are deliberately **disjoint** - each compares a different thing and is blind to what the others see:

Signal	Compares	Blind to
Staleness note in tool answers	Source files against the session	A stale binary; an incomplete analysis run
`CONFIG DRIFT` / `VERSION DRIFT`	Tool names and config-DB schema	A smarter fact producer under the same tool names
`ANALYZER DRIFT`	The server's build commit against the HEAD of the analyzed repository	Everything outside the analyzed repository

Work through them in this order when an agent reports something implausible.

`CONFIG DRIFT`**.** The active MCP profile changed since this server started - a GUI edit in the `MCP Profiles` panel, or another process writing the config DB:

```
⚠ CONFIG DRIFT: the active MCP profile changed since this server started - restart/reconnect the aicb MCP server to load the current configuration (tools, instructions, session auto-refresh mode). In-session the skill-derived instructions never refresh, the auto-refresh mode is fixed at start, and a GUI edit does not update the tool list until restart.
```

The same sentence appears once as a breadcrumb on stderr (`warning: aicb MCP config drift - ...`), because `server_info` is a rare call. **Solution:** restart or reconnect the server.

`VERSION DRIFT`**.** This binary and the config DB disagree. Two schema variants and one tool-pool variant exist:

```
⚠ SCHEMA DRIFT: the config DB is at schema v<N>, newer than this binary's latest known migration v<M> - the DB was migrated by a NEWER aicb build; this binary lags it. Rebuild/reinstall the standalone tool to catch up.
```

```
⚠ PENDING MIGRATION: the config DB is at schema v<N>; this binary ships migrations up to v<M> (<k> pending) - run the app/GUI (or a --db-path start) to migrate the DB.
```

```
⚠ TOOL POOL DRIFT: the active profile persists <k> tool name(s) this binary does not register ('<a>', '<b>', ...). Two causes look identical here and the fixes are opposite: either the DB profile was written by a NEWER build (then rebuild/reinstall the standalone tool), or the tool was RETIRED and the profile still names it (then re-save the profile in the GUI's 'MCP Profiles' panel, which rewrites the selection without it - a rebuild would only repeat this message). The entry is inert either way: an unknown name is ignored when the tool set is resolved.
```

The tool-pool check works on **tool names**; a new parameter on an existing tool is invisible to it - the build commit is the finer-grained signal. Up to 10 names are listed.

`ANALYZER DRIFT`**.** The signal the other two cannot see: the build commit of this binary against the HEAD of the repository holding the analyzed solution.

```
⚠ ANALYZER DRIFT: this binary was built from a commit <N commits BEHIND | DIVERGED from (<a> ahead, <b>
behind)> the HEAD of the repo it is analyzing (<directory>) - <k> of them <touches|touch> an analyzer project.
Facts the analyzer produces - fan-in, dead code, side effects - may be the OLDER analyzer's, including a zero
that the newer one fills. Rebuild/reinstall the standalone tool. Neither of the other two signals can see this:
the staleness trailer compares SOURCE files (unchanged here) and the version/pool check compares tool NAMES (a
smarter fact producer keeps every name).
```

There are four further outcomes, and none of them is a warning:

- `Analyzer: IN SYNC with the analyzed repo - this binary was built from the current HEAD of <directory>.`
- `Analyzer: this binary is <N commits> AHEAD of the analyzed repo's HEAD (<directory>) - built from work that repo does not have yet, so the analyzer is newer than the code, not older.`
- A behind distance where **no** intervening commit touches an analyzer project is reported quietly, with the closing sentence `The analyzer's own code is unchanged across that distance, so the facts it produces are current and this gap is no reason to rebuild.`
- `Analyzer: drift UNKNOWN - git did not answer within 2 s in <directory>, so this binary's build commit could not be compared with that repo's HEAD. ...` - nothing was measured, so this is neither a warning nor a proof of currency; call `server_info` again. When an earlier directory timed out and a later one was measured, the measured line adds `Not measured in <directory>: git did not answer within 2 s there, and a checkout of the same repository can stand at another HEAD.`

The analyzer drift line is silent without an analyzed session, on a repository that does not know this commit (any foreign solution), and whenever the binary has no commit id. It therefore stays silent in the public release builds that deliberately omit the commit.

`get_diagnostics` - the fast pre-build check

`get_diagnostics` reports the compiler errors, warnings and info messages of the session's solution - the real Roslyn build output, in seconds instead of a full `dotnet build`. It is the tool to run after `refresh_session` when you want to know whether an edit broke the build.

- **Call `refresh_session` first.** The diagnostics are compiled from the session's snapshot, not from the files on disk. Under the shipped `reactive` mode the tool is refreshed for you before it answers, so the explicit call is redundant rather than wrong - and it is the only spelling that is correct in every mode. An answer computed from a graph known to be behind the disk - including a restore since the load - leads with `verdict: 'stale'` and a `verdictReason` before any count.
- **Parameters.** `sessionId`, `severityFloor` (`auto` (default), `error`, `warning`, `info`, `hidden`; `auto` lists only the errors when there are any, and `warningsNotListed` counts the warnings it left out - ask with `warning` to list them too), `scope` (`solution` by default, or a file-path substring), and `runSuppressors` (default `false`; runs the projects' DiagnosticSuppressors, as a real build does). An unknown `severityFloor` is rejected with the valid values. `scope` filters the result, not the work: every project is compiled either way.
- `incompleteProjects`**.** A project whose compilation cannot resolve its core references is not listed in the diagnostics; it is disclosed separately, with the number of diagnostics it took with it. A freshly created, never-built worktree reports every project there - that is the honest unrestored state, not a defect. `verdict: 'inconclusive'` and `reliable: false` say so, and one `dotnet restore` or `dotnet build` makes the tool work for the rest of the session. This verdict is deliberately rare: it fires only when a strict majority of projects are incomplete.
- **Read the counts with their rules.** `incompleteRatio` and `suppressedDiagnosticsTotal` appear whenever even one project is incomplete. `suppressedDiagnosticsTotal` spans every severity and is not deduplicated, while `errorCount` counts errors only and is deduplicated across target frameworks - a total that dwarfs the visible counts means this measured a fraction of the solution.
- **Cascade diagnostics.** A project that resolves its own references but depends on an incomplete project can list `CS0012`, `CS0234`, `CS0246` or `CS0518` although it compiles fine on a real build. Such entries stay listed but are marked `cascadeFromIncomplete: true`, the response counts them as `cascadeClassifiedCount`, and each incomplete project names its `affectedDependents`. Do not read `errorCount` as "N real errors" without this.

- **Reference-binding advisories.** `CS1701` / `CS1702` ("assuming assembly reference X v1 matches X v2") are filtered out on core-resolved projects and disclosed as `referenceBindingAdvisoriesSuppressed`.
- **Restore and platform.** `unrestoredProjects` lists projects whose unresolved-name errors come with no restore output (`project.assets.json`), and its note names `dotnet restore` (or `dotnet workload restore`). `uninitializedSubmodules` names git submodules that were never initialized although the build files of a project with unresolved type or namespace names point into them - no restore or build brings those sources. `missingCompileItems` names compile items a project with errors in the answer lists without a file on disk, typically the output of a build step that did not run. `generatedCodeGaps` names code a generator or the WPF markup compiler did not write. `designTimeBuildPlatform` reports the platform decision where there is one: the platform a solution was loaded at, or why no single platform could be chosen, and a refused sidecar value.
- **Live only.** Diagnostics are not persisted, so this needs a live session. A recalled session is rejected; use `refresh_remembered` for a live one. The list is capped at 200; the counts reflect the full set.
- **Compiler diagnostics only.** Third-party Roslyn analyzer diagnostics are out of scope; `suppressors` names the analyzer references whose DiagnosticSuppressors did not run where the answer counts a diagnostic, and `runSuppressors: true` runs them.

`usage_report` - what the server actually did

`usage_report` reads the server-side tool-call telemetry: every `tools/call` invocation recorded in the config DB's `tool_calls` table. It is cross-client, not a transcript of one chat.

Read the scope first. The recording sits on the `tools/call` pipeline only. A sub-query inside `batch` or `measure` records the parent call and no child row, and a one-shot `aicb call` writes the same row under the client name `aicb-call` when it is given `--db-path`, and nothing without one. Every count is therefore a **lower bound**, and a tool at 0 was not called through this door rather than not called.

Parameters.

Parameter	Meaning	Default / limits
`topTools`	Cap on the per-tool breakdown, ranked by call count.	`30`; clamped to 1..200
`sinceDays`	Count only calls from the last N days; narrows every figure, and the applied cutoff comes back as `windowSinceIso`.	whole log; clamped to 1..3650

What it returns. Overall totals (calls, errors and error rate, distinct tools and sessions, client connections, first and last timestamp, the clients and server versions seen) and a per-tool breakdown: calls, errors, latency and result size, each as average, p50, p90 and maximum. The percentiles are the honest read for a right-skewed latency distribution, where one warmup call drags the average.

Several fields answer specific questions:

- `errorClasses` - the exception-type histogram behind a tool's errors. `McpException` is a **guided** failure the tool raised on purpose (an expired session, an unknown policy token); any other type is a defect suspicion worth chasing.
- `argumentBindingErrors` - how often a caller named an argument the tool does not have, so the call never reached the tool. This is neither a defect nor a refusal; it shows where a tool's argument surface confuses callers.
- `ignoredArgumentCalls` - per tool, how often a call answered although it sent an argument the tool does not have, so that argument had no effect.
- `clientSessions` - how many client connections the calls came from; `distinctSessions` counts session references, and every chat that passes the same solution path shares one.
- `aliasApplied` - how often a tool was called with an accepted alias instead of the real parameter name.
- `protocolVersions` and `clientEras` - the share of calls per protocol revision, and an (era × client) cross-tab. A null protocol version means the row predates this instrumentation (also reported as `preInstrumentationCalls`) and belongs in the denominator; a null client name means the client did not identify itself. `clientEras` is capped at 50 groups.
- `facets` - how often each task facet was addressed. It is omitted until a call has addressed one.

- `poolCoverage` - `poolSize`, `poolToolsFired`, `poolToolsNeverCalled`, `outOfPoolCallsIncluded` and `outOfPoolTools`. Read coverage here, **not** from `distinctTools`: that figure counts every tool ever called, including tools outside the current pool and tools from earlier pools, so holding it against the pool size overstates coverage. And a name in `poolToolsNeverCalled` was never called **top-level**, which is not the same as never called.

When telemetry is unavailable, the answer is `available=false` with the note:

```
No telemetry available - the server is running DB-free, or the tool_calls table is not readable.
```

Privacy note. Exactly one telemetry field carries free text: the message of a failed exception. Set `AICB_MCP_ERROR_TEXT=off` and only the exception **type** is recorded. The error message is capped at 500 characters. Deleting the recorded data is a GUI action (the `MCP Usage` panel); there is no CLI verb for it.

`list_skills`, `list_mcp_profiles`, `docs()` and `aicb call`

- `list_skills` is the capability map: every tool appears once in the pool groups (`inPool` or `outOfPool`). `uncatalogued` is a separate name-based section, not a third pool group; its tools can still be in either pool under an override. No profile menu names them, so only `AICB_MCP_TOOLS` can expose them.
- `list_mcp_profiles` lists the profiles, their ids, their tool selections and which one is active. It needs a config DB (see above).
- `docs()` is the server's built-in operating manual, with pages for installation, first context, tool navigation, solution configuration and the glossary.
- `aicb call <tool>` invokes one MCP tool once from a terminal, without a client - useful to reproduce what an agent reported. Example: `aicb call server_info`. It writes to the usage telemetry only when it is given `--db-path` (under the client name `aicb-call`); without one it records nothing. Long analyses print progress to stderr as `progress: ...`, so stdout stays clean for the result.

12.12 What the server writes to stderr at startup

These lines land in your MCP client's stderr log. The server also prints a tool-curation line on every start and warns when `aicb.mcp.json` is invalid. The pinned-profile error is the only case below in which the server does not start.

Line (core)	Meaning
`No --db-path given; using the standard config DB (same as the GUI): <path>`	Normal case, not a warning.
`warning: MSBuild could not be registered (<reason>). analyze_solution/refresh_session will fail; the remaining tools stay usable.`	No MSBuild on the machine. The session tools fail; the rest works.
`warning: standard config DB path could not be resolved (<reason>); starting DB-free (default profile's pool).`	No profile applies, but the server runs.
`warning: --db-path not found (<path>); starting without a profile (default profile's pool).`	A typo in the path.
`warning: --db-path schema migration failed (<reason>); starting without a profile.`	The database exists but cannot be migrated.
`warning: MCP profile resolution failed (<reason>); starting without a profile.`	The profile could not be read.
`warning: ... aicb.mcp.json ...`	The optional config file was unreadable or invalid; defaults remain in effect.
`MCP tool curation: ... active MCP profile=...; session auto-refresh=...`	The resolved tool pool/profile and auto-refresh mode for this process.

Line (core)	Meaning
`error: unknown MCP profile '<id>' (--mcp-profile). Available: <list>.` followed by `error: --mcp-profile '<id>' could not be applied (see above). Refusing to start on a different profile.`	**The only case in which the server does not start** (exit code 1). An explicitly pinned profile is never silently replaced by a different one.
`Pinning the MCP profile to '<id>' for this process (--mcp-profile); the profile persisted in the config DB is left unchanged.`	Confirmation, not a warning.
`Active MCP profile: '<name>' (<id>)[ pinned via --mcp-profile; not persisted]; tool set: <spec or (lean)>; skill: <set or (none)>.`	Confirmation of what this process serves.

Rule of thumb. The MCP server almost always starts - it prefers to degrade and say so. The one exception is an explicitly pinned, unknown profile.

12.13 Server-side guards and limits at a glance

The guards, in the order a call passes them:

1. **Load gate.** A call for a solution that is currently loading waits up to `AICB_MCP_LOAD_WAIT_MS` (default 45 s) and then reports what is holding it instead of blocking mutely.
2. **Profile guard.** A tool outside the active profile is refused with the message in "Tool `<name>` is not in the active profile". The same rule applies per sub-query inside `batch` and `measure`.
3. **Argument binding.** A missing required argument or an unknown argument name produces the binding-failure message; a dropped optional argument is disclosed on the successful answer as `ignored-arguments`.
4. **Tool failure.** Any other exception from a tool is wrapped so that its type, cause and remedy reach you instead of the bare "An error occurred invoking '". `McpException` messages and cancellations pass through unchanged. The generic remedy is `Retry the call (optionally after refresh_session, which rebuilds the analysis); if it persists, the cause above is the thing to report.`
5. `dbPath` **guard.** Tools that need a database reject an empty path and a path that names a directory, as described above.
6. **Zero-hit steer.** A symbol query that resolves to nothing adds a `nearest` suggestion naming the closest declared symbol, so a typo is distinguishable from a real empty result. Tools that report a resolution kind use `resolvedKind: "not_found"` for the same purpose. A tool whose empty answer is a legitimate measurement (for example a complexity ranking with nothing above the floor) does not get a suggestion.

Numeric limits you may meet in practice:

Limit	Value
Registered tools / default profile / `Full Select`	82 / 54 / 72 tools
`batch` and `measure` sub-queries per call	16
`batch` response budget	~9,000 tokens
Read-only tools dispatchable through `batch` and `measure`	48
Slice-tool budget (`get_context` , `explain_symbol` , `pack_for_task` , `prepare_task`)	default ceiling 10,000 tokens; an explicit `budget` is floored at 8,000
Host inline rendering limit (a larger slice is written out instead of returned inline)	~50,000 characters
`get_diagnostics` list	capped at 200
`usage_report` `topTools` / `sinceDays`	default 30, clamped 1..200 / clamped 1..3650
Incomplete-run alarm: project names spelled out	5
Dropped-argument notice: names listed / characters per name	8 / 64

Limit	Value
Failure message: cause and parameter inventory	400 characters each
Telemetry error message	500 characters
Session TTL / sessions held / sweep interval	90 minutes / 8 / 5 minutes
Staleness check window (how often the source check runs)	5 seconds
Background-refresh debounce	15 seconds
Registry load-gate wait	45 seconds
`tools/list` cache lifetime	15 minutes

The three timing values have environment-variable overrides: `AICB_MCP_STALENESS_WINDOW_MS`, `AICB_MCP_DEBOUNCE_MS` and `AICB_MCP_LOAD_WAIT_MS`. Each takes milliseconds; a zero, a negative or an unparseable value falls back to the built-in default rather than switching the behavior off.