

AICONTEXTBUILDER

AICB - General Documentation

Describes AICB V0.5.505.1

As of 2026-10-26

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

Contents

1 Introduction

1.1 What AIContextBuilder is
1.2 The problem it solves
1.3 Who it is for
1.4 The three faces
1.5 How AICB differs from text search
1.6 How AICB differs from a language server
1.7 What AICB does not do
1.8 How this manual is organized

2 License, installation and updates

2.1 The license
2.2 Distribution channels
2.3 Prerequisites
2.4 Installing the .NET tool (CLI and MCP server)
2.5 Installing the desktop app on Windows
2.6 One installation per machine
2.7 The Windows SmartScreen warning
2.8 Verifying your download
2.9 Updating
2.10 Uninstalling
2.11 Removing all data
2.12 Support and security reports
2.13 Building from source

3 Core concepts

3.1 Solution
3.2 Project
3.3 Namespace, type and member
3.4 Unit
3.5 Analysis
3.6 The tree
3.7 Facts
3.8 Semantic axes and provenance
3.9 The `<ai>` annotation
3.10 Edges
3.11 Fan-in and impact
3.12 Dead code and cycles
3.13 Layers
3.14 The context document
3.15 Sessions and snapshots
3.16 Insights and severity

4 How the analysis works

4.1 What goes in

4.2 Opening a solution
4.3 The analysis run
4.4 Incomplete projects
4.5 Multi-targeting
4.6 How long it takes
4.7 Side-effect classification
4.8 Layer profiles and architecture analysis
4.9 Dead-code detection
4.10 Circular namespace dependencies
4.11 Test detection and coverage gaps
4.12 Dependency injection resolution
4.13 PageRank and the relevance score
4.14 Metrics

5 The context document (AI-Builder-MD)

5.1 The file
5.2 The two notations
5.3 The document header
5.4 The sections at a glance
5.5 The architecture graphs
5.6 DOMAIN, ENUM_SUMMARY and FILE_INDEX
5.7 BUILD_AND_UI_FILES
5.8 AUXILIARY_FILES
5.9 The solution tree
5.10 The type blocks
5.11 Method compression modes
5.12 The legends
5.13 Detail levels and auto-compression
5.14 Token budget and trimming
5.15 Document modes
5.16 What the document does not tell you
5.17 A complete example

6 Templates, rendering and markup analysis

6.1 What a template controls
6.2 Built-in templates
6.3 Detail presets
6.4 MD profiles
6.5 Tag schemas
6.6 Expansion strategies
6.7 Compression rules and pipeline profiles
6.8 How a document is assembled
6.9 Markup analysis: XAML and AXAML

7 Profiles, master data and solution configuration

7.1 Master data and profiles
7.2 Per-solution configuration
7.3 The `.aicb.json` sidecar
7.4 How settings are resolved

8 Insights: the code quality catalog

8.1 What an insight is
8.2 Severity levels
8.3 Categories
8.4 The insight catalog
8.5 Quality profiles
8.6 Suppressing and dismissing findings
8.7 The technical debt estimate
8.8 Known limits of the analysis

9 Command-line reference

9.1 Invocation, help and version
9.2 Exit codes
9.3 `aicb init`
9.4 `aicb analyze`
9.5 `aicb export`
9.6 `aicb import`
9.7 `aicb list`
9.8 `aicb mcp`
9.9 `aicb call`
9.10 Environment variables
9.11 Typical headless workflows

10 Data, storage and privacy

10.1 Sessions and snapshots
10.2 The database file
10.3 Backups
10.4 Switching and creating databases
10.5 Import and export
10.6 Where everything lives on disk
10.7 Removing your data
10.8 What the product does with your data
10.9 API keys
10.10 Local MCP usage recording
10.11 Third-party components and licenses
10.12 What the product promises and requires

11 Troubleshooting

11.1 Startup and installation
11.2 Analyzing a solution
11.3 LLM and network errors
11.4 Data and persistence
11.5 Diagnostic means
11.6 Support

12 Glossary

12.1 A - B
12.2 C
12.3 D - G
12.4 H - L

12.5 M - N
12.6 O - R
12.7 S
12.8 T - Z
12.9 Terms that are easy to confuse

1 Introduction

1.1 What AIContextBuilder is

AIContextBuilder (`aicb`) is a Roslyn-based tool that turns a C#/.NET solution into dense, structured Markdown for LLM consumption, and exposes the same analysis engine as an MCP server so that coding agents can navigate your code semantically instead of by text search.

Roslyn is Microsoft's .NET compiler platform - the engine behind the C# compiler and the C# language services in editors and IDEs. AICB uses it to build a symbol graph of your solution (types, members, references, inheritance, interfaces, dependency injection registrations) and to derive facts from that graph: who calls what, what a change would affect, where a method is implemented, which methods touch the file system or a database, how complex they are, which tests cover them. Those facts answer questions on their own, and they feed the Markdown context documents that AICB renders from the same analysis.

The tool prints its own one-line description:

```
aicb - Roslyn-based .NET Markdown context generator for AI/LLM workflows. (c) Gregor Dadera - free
for private, hobby and educational use and qualifying organizations; see LICENSE.txt.
```

AICB is not an IDE, a build tool or a refactoring tool. It reads your solution and reports on it; it produces no assemblies and does not execute your application code. Two boundary cases keep that read from being a passive one, and both are documented in "How the analysis works": diagnostics are computed from in-memory Roslyn compilations, and opening a solution runs its MSBuild design-time logic. The one exception to read-only behavior is the desktop app's Details tab, which can edit and save a source file - everywhere else, AICB only tells you things.

1.2 The problem it solves

`grep` finds substrings. It does not know what a symbol is. A search for a type name returns every line that happens to contain that text, and it misses the places where the compiler - not the text - creates the reference. Before an edit, an agent (or you) needs different answers: who calls this, how far does a change reach, where is this implemented, what does this method touch, and how much context does this task need.

AICB answers those questions from the symbol graph. The difference is not cosmetic: in the current measurement, a search that returns 153 line positions where 40 consumer types are meant is harder for a model to use than the aggregated answer, and a reference that exists only in the compiler's view is invisible to a text search entirely. The sections "How AICB differs from text search" and "How AICB differs from a language server" below spell this out with the measured numbers.

One convenience is worth knowing from the start: every tool that takes a session also accepts the absolute path to a `.sln`, `.slnx` or `.slnf` file directly and analyzes it on first use. There is no separate preparation step between you and an answer.

1.3 Who it is for

Three groups of users are visible in the product today, in the order in which it serves them:

1. **Developers who drive a coding agent on a C#/.NET solution** (Claude Code, Cursor, Cline, Codex, OpenCode). This is the primary audience: `aicb init` and `aicb mcp` exist for it, and the agent skill that `aicb init` installs teaches an agent when to reach for which tool - and when plain `grep` is still the right choice.
2. **Developers at the shell**, who ask a single semantic question with `aicb call` or generate a context document with `aicb analyze` and paste it into a chat.
3. **Desktop app users**, who curate what goes into an exported context document - run templates, detail presets, Markdown profiles, quality profiles - and configure how a solution is analyzed: namespace exclusions, test detection, layer mapping, and the MCP profiles a client sees.

A related group gets the quality and architecture half: `solution_metrics` with a gate expression (`aicb analyze --fail-on`), layer profiles with violation checks, cycle detection, a technical-debt estimate and `list_insights` are useful to whoever owns code quality. Note that there is no team feature behind them - no shared server and no multi-user state; see "What AICB does not do" below.

License

AICB is closed source. The license is free of charge for natural persons (private, hobby and educational use), for accredited educational institutions (teaching, learning and non-commercial research), and for organizations that reach none of three thresholds:

- 100 employees,
- EUR 10 million annual turnover,
- 21 developers.

A subsidiary counts together with enterprises under common control, and the thresholds measure your organization only - the size of your clients does not matter, including when you work on their premises. Once your organization reaches or exceeds one threshold, a 90-day contractual transition period begins; use stays free during that period and a commercial license is required to continue afterwards. The period is text in the agreement, not a product timer: AICB sends no threshold or deadline data, blocks no feature and does not technically stop. Commercial terms are agreed individually; licenses start at EUR 25 per licensed developer per month.

Connecting AICB to MCP clients, agent harnesses, scripts, build systems and CI through the documented interfaces is permitted. Redistribution, modification, repackaging, resale, hosted third-party access and competing products are not. The software contains no license server, activation token or outbound licensing telemetry; local MCP tool-call usage is not license telemetry. `EULA.md` ships with the product as the full bilingual agreement and `LICENSE.txt` as its summary; both are also published in the public repository github.com/gregordadera/aicb-roslyn-mcp. The chapter "License, installation and updates" has the details.

1.4 The three faces

AICB has three faces, and they are not three independent programs: all three run the same Roslyn analysis and the same rendering engine.

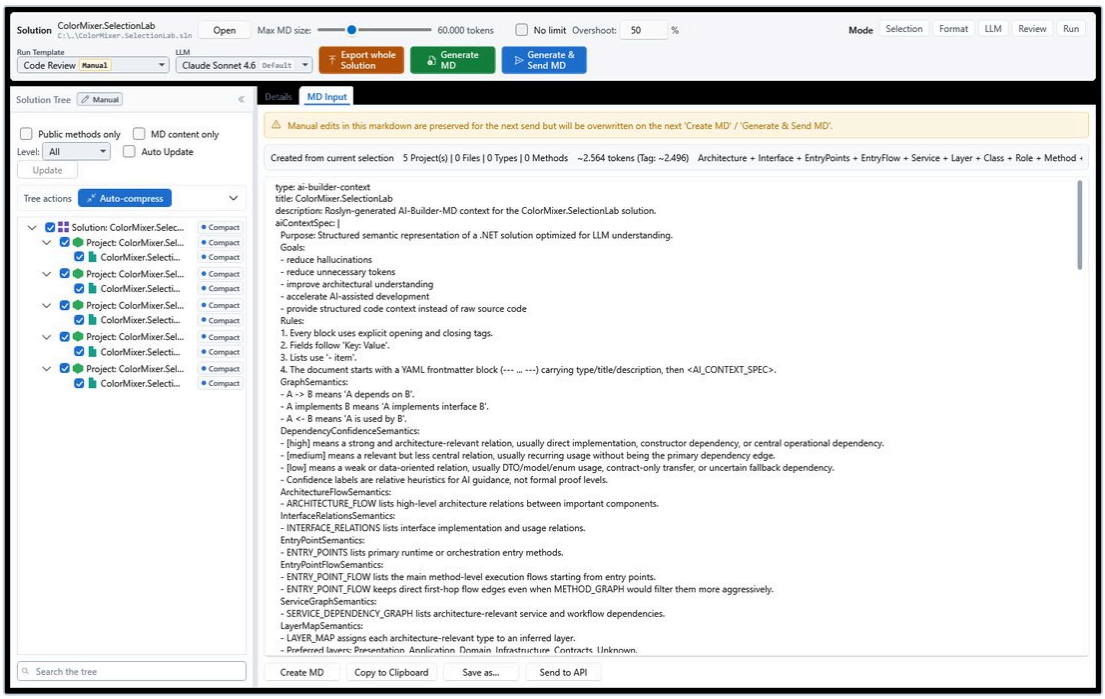
Face	What it is	Where it runs
The desktop app	The graphical curation and configuration surface - and the only face that can edit a source file	Windows only
The command line (`aicb`)	Seven verbs for analysis, export, import, listing, wiring and the MCP server	Windows, Linux, macOS

Face	What it is	Where it runs
The MCP server (`aicb mcp`)	A stdio JSON-RPC server that offers the semantic tools to a coding agent	Windows, Linux, macOS

The MCP server and the CLI install from nuget.org; the desktop app downloads from GitHub Releases as a Windows installer or a portable ZIP.

The desktop app

The desktop app is where you load a solution, browse it in a tree, and read the context document AICB renders from it. It is also where the master data lives: run templates, detail presets, Markdown profiles, quality profiles and MCP profiles are managed here, together with the settings that decide how a solution is analyzed. It is the only face that can edit a source file - the Details tab opens a file in a real editor and writes it back with Save or `Ctrl+S`, keeping its original encoding and line endings; a file that is missing or fails to load stays read-only. The desktop app can also send a rendered context to an LLM endpoint you configure - that is what its "Send to API" button does, and the endpoint may be a local model.



The desktop app with a solution loaded: the solution tree on the left, the generated context document on the right

The command line

`aicb` is a .NET global tool for Windows, Linux and macOS:

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

It has seven verbs:

Verb	What it does
`aicb init`	Wires AICB into this project for an MCP client: adds the `aicb` entry to `.mcp.json`, writes the agent skill to `.claude/skills/`, and installs the symbol guard for the agent harnesses the project uses. Existing files are kept unless you pass `--force`.
`aicb analyze`	Runs Roslyn analysis on a solution and emits context Markdown.
`aicb export`	Re-renders Markdown from an existing session database, without a Roslyn re-run.
`aicb import`	Imports a constellation JSON into a target database (templates, presets, constellations).
`aicb list`	Lists built-in and custom master entities; `run-templates`, `detail-presets` and `model-profiles` are three of fourteen kinds.
`aicb mcp`	Starts the MCP server (stdio JSON-RPC) for Claude Code, Cursor, Cline and other MCP clients.
`aicb call`	Invokes one MCP tool once and prints its result - no server, no client needed.

Run `aicb <command> --help` for the options of a verb.

The MCP server

`aicb mcp` starts an MCP (Model Context Protocol) server over stdio JSON-RPC. It is a verb of the same `aicb` command, not a separate program: installing the CLI installs the server.

By default the server exposes a lean pool of 54 tools covering all nine task facets - General, Exploration, Refactoring, Debugging, Review, Testing, Documentation, Architecture and Performance. The unnarrowed pool of 72 tools is one profile switch away: `--mcp-profile mcp-profile/full` at the command line, or the "Full Select" profile in the desktop app. The environment variable `AICB_MCP_TOOLS` is an operator override; `AICB_MCP_TOOLS=all` exposes every tool the assembly registers, including the opt-in infrastructure tools. One binary serves both current MCP protocol revisions over stdio - the `2025-11-25` handshake and the `2026-07-28` stateless entry point - so clients on either revision connect.

Most tools need an analyzed solution. Every tool that takes a session also accepts the absolute path to a `.sln`, `.slnx` or `.slnf` file directly and analyzes it on first use, so a client can start asking without a preparation step.

With `--db-path` the server reads the per-solution configuration and the active MCP profile from the configuration database you name; `--mcp-profile <id>` pins the profile for this server process only, without writing anything back. Without `--db-path` it resolves the standard configuration database - the same one the desktop app uses.

How the three relate

- **One engine.** The desktop app, the command line and the MCP server share the same Roslyn analysis and the same render path. A run template you create in the desktop app can drive a headless export with `aicb analyze --run-template` or `aicb export --run-template`.
- **The MCP server is a verb of the CLI.** There is no separate server executable, and no separate server installation.

- **The desktop app and the MCP server share one configuration database by default** - on Windows `%APPDATA%\AIContextBuilder\user-data\aicb.acb`, elsewhere under the platform's application-data folder. MCP profiles, per-solution namespace exclusions, the test-detection profile and the layer profile are edited in the desktop app and read by the server. The sharing is per machine, not per team.
- `aicb init` **wires a project for a client.** It adds the `aicb` entry to `.mcp.json`, writes the agent skill to `.claude/skills/`, and installs the symbol guard into the agent harness the project uses. For Codex and OpenCode, which do not read `.mcp.json`, it also registers the server in `.codex/config.toml` or `opencode.json` and writes the skill to `.agents/skills/`. It never overwrites an existing file unless you pass `--force`, so re-running is safe; `aicb init --hooks none` installs no guard, and `--hooks` also accepts `auto` (the default), `all`, or one of `claude-code`, `codex` and `opencode`. The guard is not a hint: it refuses a C# symbol search aimed at a type or member name and points at the AICB tool that answers it. One step stays with you, and `aicb init` names it in its output: Codex reads a project's `.codex/config.toml` only once you have trusted that project in Codex.
- `aicb call` **invokes exactly one MCP tool** - in-process, without a server or a client - and prints the result. It applies no profile filter: it can reach any registered tool, including tools outside the active pool and tools that write configuration. Use it deliberately.

1.5 How AICB differs from text search

Both `grep` and AICB can tell you that a name occurs. The difference is what each one understands by "occurs". Three examples show where the symbol graph sees what a text search cannot:

- **Compiler semantics beyond tokens.** In four test classes, the interface names `ICodeAnalyzer` and `IFileSetHasher` appear nowhere as text: the binding is created by an implicit reference conversion at the argument position of a method call. A text search finds zero occurrences of the names; AICB reports the four classes as consumers, because the compiler does.
- **Markup.** In a WPF application, XAML references such as `{x:Type ...}` and `{x:Static ...}` create dependencies that a text search sees only as text and a Roslyn-only tool does not see at all. In AICB's differential benchmark, four of the twenty cases where AICB found a dependency and the language server did not came from markup.
- **Questions a text search cannot formulate.** The transitive fan-in closure; the split between production and test consumers (`productionImpactCount`); classes of side effects (I/O, network, database, serialization, logging, cache, messaging); namespace dependency cycles; dependency injection registrations; complexity and coupling metrics.

Where text search stays the right tool: string literals, comments, log messages, config keys, version strings, and files that are not C# - `.json`, `.csproj`, `.md`. AICB routes those questions to text search rather than guessing from the symbol graph.

1.6 How AICB differs from a language server

The strongest alternative to AICB is not `grep` but a language server - in this comparison, the Roslyn language server, which has been available inside agents such as Claude Code since December 2025 and ships free. AICB's differential benchmark was run against exactly that alternative, on AICB's own solution (13 projects, about 200,000 lines of C#), with a deliberately chosen sample of twelve symbols. It is a structural comparison, not a general benchmark of answer quality - and it is explicit about where the language server wins.

What a language server does better:

- Point questions - "where is X defined", "who calls X" - are built into the agent, free, and answer the common case.
- It sees unsaved editor buffers; AICB reads from disk.
- It returns navigable positions; AICB's answers are mostly symbol-level.
- It does real fuzzy matching; `find_symbol` matches case-insensitively by substring.
- It reports analyzer diagnostics; AICB reports compiler diagnostics only.
- It is incremental; AICB needs `refresh_session` after an edit.

What a language server structurally cannot do:

- Transitive closure - it reports one level; AICB reports the closure.
- Test/production separation - for the protocol, a reference is a reference.
- A risk judgment for a change.
- Test coverage - it has no concept of a test.
- Code metrics - complexity, coupling, lines of code.
- XAML.
- State an honest negative: an empty answer from a language server is indistinguishable from the client having forgotten to ask; AICB's tools write their negatives into the answer, for example `implementations: (none)`.

The single number that shows the difference, measured against AICB 0.5.464.33: for the enum `RunType`, AICB reports **40 direct consumers and 1,064 transitive ones**, 516 of them in production code; the language server reports individual reference positions instead of the aggregated consumer graph.

Aggregation is the other half of it. Re-running the AICB side against 0.5.464.33 while retaining the benchmark's language-server payload, `find_usages` answered with 1,079 bytes covering 40 types, where the reference list answered with 41,237 bytes covering 153 positions - about **38 times more compact**. The two answers serve different purposes: an editor needs positions to click through, while a language model benefits from aggregation by consumer type.

1.7 What AICB does not do

AICB is deliberately narrow. Knowing the limits saves time:

Limit	What it means
C#/.NET only	Roslyn is the foundation. C# and .NET projects are understood; XAML/AXAML markup is read for bindings and resources. Other languages are out of scope.
Windows-only desktop app	The `aicb` command line and the MCP server run on Windows, Linux and macOS. The desktop app runs on Windows only.
stdio only	The MCP server speaks stdio JSON-RPC. There is no HTTP/SSE transport.
No multi-user state	The configuration database is a local SQLite file on your machine. Apart from client-wiring files created by `aicb init` (such as `.mcp.json` and agent-specific hook or settings files), the only AICB analysis/configuration artifact intended to travel with your repository is the git-tracked `<Solution>.aicb.json` sidecar next to the `.sln`; it carries configuration, not results.
No live editor buffer	AICB analyzes the files on disk. Unsaved changes in your editor are invisible until you save; a session that has already analyzed a solution re-reads it with `refresh_session`.

Limit	What it means
Compiler diagnostics only	`get_diagnostics` reports compiler errors and warnings. It does not run your third-party analyzers - except, on request (`runSuppressors: true`), the DiagnosticSuppressors a real build applies to compiler diagnostics - and diagnostics that a source generator reports itself are not included, so "0 errors" is not the same as "clean in the sense of your IDE".
Symbol-level answers	Most tools name the declaring type and member rather than a file-and-line position you can click. For position-level navigation, use your IDE or a language server.
No fuzzy search	`find_symbol` matches case-insensitively by substring. A typo returns no hits; a `nearest` suggestion names the closest declared symbol.
Diagnostics are not cached	`get_diagnostics` compiles on every call. Its `scope` parameter makes the answer smaller, not the work.
No outbound network in headless surfaces	The CLI and the MCP server have no outbound network capability at all. The desktop app can contact only an LLM endpoint you configure: for a run, a model-profile connection test, or the first-load Layer Profile / Exclude Namespaces proposal when those auto-init flags are armed.
A local call log	The MCP server records the tool calls it handles in your configuration database - that is what `usage_report` and the desktop app's usage panel read. The log stays on your machine and nothing is transmitted.

Note: opening a solution runs its MSBuild build logic to resolve references, and building its compilation runs the source generators its projects reference - exactly as Visual Studio or `dotnet build` does. Analyze only solutions you trust.

1.8 How this manual is organized

This manual is the General volume. It covers the product itself, the `aicb` command line, and the configuration that all three faces share. Two companion volumes go deeper: "MCP server" documents the tools, profiles and session model that an agent sees, and "Desktop app" documents the desktop application's windows, panels and settings. Where a topic belongs to one of those volumes, this manual refers to it by title instead of repeating it.

2 License, installation and updates

AIContextBuilder is closed source: you install a released build, and there is no source code to compile. This chapter covers the license terms, the channels the product ships through, what your machine needs before it can analyze anything, how to install and update each form, and how to uninstall it and remove every file it has written.

2.1 The license

AIContextBuilder is licensed under the AIContextBuilder End-User License Agreement (EULA). The full bilingual agreement `EULA.md` and its non-binding orientation summary `LICENSE.txt` are shipped with every distribution form and packed into the NuGet package. The public current text is https://github.com/gregordadera/aicb-roslyn-mcp/blob/main/EULA.md; the immutable reference for version 0.6 is https://github.com/gregordadera/aicb-roslyn-mcp/blob/eula-v0.6/EULA.md. The German version is binding for natural persons habitually resident in Germany and organizations with their registered office or principal place of business in Germany; the English version is binding for other licensees unless an individual agreement selects the German version.

Free use below three thresholds

Use is free of charge for as long as your organization reaches **none** of these thresholds:

Threshold	Value
Employees	100
Annual turnover	EUR 10,000,000
Developers	21

Reaching or exceeding even one of them starts the commercial-license transition described below. "Organization" means you together with enterprises under common direct or indirect control: a subsidiary is counted together with its group. The control tests in Article 3(3) of EU Recommendation 2003/361/EC determine linked enterprises. A minority participation without control does not by itself join two organizations, and the thresholds in that recommendation do not apply; only the three values above are authoritative.

How the terms are counted:

Term	Counted as
Employees	all persons working for the organization, regardless of the form or extent of their engagement; counted as headcount, not as full-time equivalents
Developers	every natural person who, in the course of their work for the organization, writes, modifies or reviews source code, regardless of job title, form or extent of engagement. People who only plan, coordinate or test software without reference to source code are not counted. Automated systems and LLM agents are not natural persons and do not count as additional developers
Annual turnover	the consolidated turnover of the organization in its most recently completed financial year; foreign currencies are converted at the European Central Bank reference rate on the balance sheet date. For organizations without turnover in the commercial-law sense, this condition is disregarded
Assessment date	the end of each financial year, and any other point at which documented facts show that a threshold has been reached or exceeded

Always free of charge, regardless of the thresholds:

- **Natural persons** using the software privately, as a hobby, or for education.
- **Accredited educational institutions** for teaching, learning and non-commercial research. Commercial activities of the institution, in particular contract research, fall under the normal rules.
- **Work for clients:** the thresholds measure your organization only. The size of your clients does not matter, including when you work on their premises.
- **Public bodies** fall under the normal thresholds; there is deliberately no separate clause for them.

When a commercial license is required

If your organization reaches or exceeds one threshold, a **90-day contractual transition period** begins. Use stays free of charge during that period; a written commercial license is required to continue afterwards. Commercial licenses start at EUR 25 per licensed developer per month; terms are agreed individually, and price and scope can depend on licensed users, requested support and response scope, and agreed priority or delivery commitments for improvement requests. A commercial agreement may include support with defined response and security-fix targets (for example, answers within two business days and fixes for confirmed vulnerabilities within ten business days), version maintenance, prioritized general product improvements and source-code review under NDA; payment alone creates no unstated SLA or implementation promise. Contact **aicb@dadera.de**.

The 90 days are contractual text only. The software starts no timer, transmits no threshold or deadline data, blocks no feature and does not technically stop when the period ends. There is no license server, activation token, watermarking or outbound licensing telemetry. The MCP server can record tool calls locally for `usage_report` and the desktop app's MCP Usage view; these data are not used for licensing and never leave the machine. Compliance is your responsibility.

Customer code is never collected or used for product improvement automatically. Any examination requires material or access deliberately provided by the customer and a separate agreement on scope, confidentiality and retention. Accepted improvements are developed as general AICB capabilities rather than as a customer-specific fork unless an individual agreement says otherwise.

Donations

Donations are voluntary and welcome, for example via https://github.com/sponsors/gregordadera, but they are a separate matter. A donation does not replace a commercial license where one is required, no matter its size, and it does not create a claim to support, updates or bug fixes.

Restrictions

You may not:

- reverse-engineer, decompile or disassemble the software, except where mandatory applicable law expressly permits it (for interoperability, German Copyright Act sections 69d and 69e);
- remove, alter or obscure copyright notices, trademarks or version information;
- redistribute, rent, lease, resell or otherwise make the software available to third parties;
- modify or translate the software, or create derivative works from it;
- bundle or embed the AICB binaries into another product, or offer them as a hosted/shared service for direct third-party use;
- present or distribute the software under your own name or brand, or as your own work (re-branding);
- build competing products on the basis of the software.

Backups for your own use are permitted. Connecting AICB to MCP clients, agent harnesses, scripts, build systems and CI/CD through its documented interfaces is also permitted, as is internal use by consultants serving clients. You retain all rights in your source code and may use, adapt, version and share generated context, exports, `.aicb.json` sidecars and files installed by `aicb init`, subject to rights in their underlying material and third-party content.

"**AIContextBuilder**" and "**AIContextBuilder for .NET**" are product names used by Gregor Dadera; no registration is claimed. Factual references (for example "created with AIContextBuilder") are fine; re-branding or implying a partnership is not.

Warranty, liability and applicable law

The software is provided "as is", without warranty. Liability is unlimited for intent, gross negligence, and injury to life, body or health; for ordinary negligence it is limited to the breach of essential contractual duties and capped at typical, foreseeable damage; otherwise it is excluded. Liability under the German Product Liability Act remains unaffected. German law applies, excluding the UN Sales Convention (CISG). For merchants and legal entities the exclusive place of jurisdiction is the domicile of the licensor; for consumers the statutory rules apply. By downloading, installing or using the software you accept the EULA.

2.2 Distribution channels

AIContextBuilder comes in three forms, all built from the same analysis engine:

Form	Contains	Platforms	Where to get it
.NET tool (NuGet package `aicb-roslyn-mcp`)	CLI and MCP server	Windows, Linux, macOS	nuget.org, installed with `dotnet tool install -g aicb-roslyn-mcp`
Windows installer	Desktop app, CLI and MCP server	Windows 10 or later, 64-bit	GitHub Releases: `AIContextBuilder-Setup-<version>.exe`
Portable ZIP	The same payload, no installation	Windows 10 or later, 64-bit	GitHub Releases: `AIContextBuilder-<version>-win-x64.zip`

All downloads are at https://github.com/gregordadera/aicb-roslyn-mcp/releases.

The desktop app is Windows-only. On Linux and macOS you get the CLI and the MCP server through the .NET tool; there is no GUI build for those platforms. The Windows builds are 64-bit; the installer also runs on ARM64 Windows through x64 emulation.

Note: The public repository carries the documentation, the license and the releases, not the source code.

2.3 Prerequisites

Requirement	Applies to	Why
MSBuild (a .NET SDK or a Visual Studio installation)	every form	Roslyn loads a solution through MSBuild; without it no solution can be analyzed
.NET 10 SDK	the .NET tool	needed to install and run the tool; the same SDK also provides MSBuild. Versions up to 0.5.465.11 needed the .NET 8 SDK
Windows 10 or later, 64-bit	installer and ZIP	the desktop app is a WPF application

Requirement	Applies to	Why
Administrator rights	the installer	it installs under `Program Files` and maintains the machine-wide `PATH`
none	the ZIP	unpack and run

Both Windows forms are self-contained: they carry their own .NET runtime, so no .NET installation is required to start them. This does not remove the MSBuild requirement. Opening a solution runs MSBuild's design-time build, exactly as Visual Studio or `dotnet build` does.

Note: If MSBuild cannot be found, `aicb analyze` exits with code 5. The MCP server (`aicb mcp`) and `aicb call` print a warning and start anyway, because their session-less tools keep working. The desktop app checks for MSBuild before anything else, shows `MSBuild not found` and exits.

Note: The analysis runs with server garbage collection limited to four heaps. On a machine with little memory you can override this with the environment variable `DOTNET_gcServer=0`, which takes precedence over the shipped setting.

2.4 Installing the .NET tool (CLI and MCP server)

1. Install the .NET 10 SDK if you do not have it. Without .NET 10 the tool runs on the next newer .NET on the machine and needs that version's SDK; a machine that has only .NET 8 or .NET 9 cannot start it.
2. Run:

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

3. Confirm the installation:

```
aicb --version
```

The tool is installed for your user account, normally under `%USERPROFILE%\.dotnet\tools`, and `aicb` is available on your user `PATH`. The MCP server is not a separate program: `aicb mcp` is a verb of this same command, and `aicb init` wires it into a project.

Update later with:

```
dotnet tool update -g aicb-roslyn-mcp
```

Up to 0.5.465.1 the package was called `AIContextBuilder`. An update does not cross that rename: remove the old package with `dotnet tool uninstall -g AIContextBuilder` first, then install `aicb-roslyn-mcp` as above.

Remove it with:

```
dotnet tool uninstall -g aicb-roslyn-mcp
```

2.5 Installing the desktop app on Windows

The installer

1. Download `AIContextBuilder-Setup-<version>.exe` from the release page.
2. Run it. Windows asks for administrator rights, because the setup installs under `Program Files` and maintains the machine-wide `PATH`.
3. Pick the language (German is preselected, English is available) and, if you want, a different install folder. The default is `C:\Program Files\AIContextBuilder`.
4. Read and accept the license text to continue.
5. On the options page: - `Add the "aicb" command-line tool to PATH (required for the MCP connection used by AI agents)` - selected by default. Keep it if an MCP client should be able to start `aicb` by name; clear it if you prefer to leave the `PATH` untouched. The entry is machine-wide, for all users. - An optional desktop icon - not selected by default. A Start menu entry `AI Context Builder` is always created. - If a global .NET tool is already installed, an extra group appears: `AICB is already installed as a .NET tool (NuGet):`, with the option `Remove the .NET tool - the MCP server then comes from this installation and is updated with the desktop app (recommended)`, selected by default. See "One installation per machine" below.
6. Finish the setup; you can launch the app directly from the last page.

The setup installs both executables, `gui\aicb-ui.exe` (the desktop app) and `cli\aicb.exe` (the CLI and MCP server), the full agreement `EULA.md`, its summary `LICENSE.txt`, and `THIRD-PARTY-NOTICES.txt` into the install folder.

Note: A change to the `PATH` reaches only processes started afterwards. Consoles, editors and AI agents that are already open find the `aicb` command only after a restart; the installer's last page says so.

Note: If the .NET tool cannot be removed because it is still running as an agent's MCP server, the setup reports this and offers `Retry` after you close the agent. If you cancel, it prints the command to run later.

The portable ZIP

1. Download `AIContextBuilder-<version>-win-x64.zip` and extract it. Extracting with "Extract Here" produces one versioned folder, `AIContextBuilder-<version>-win-x64`.
2. Start the desktop app by double-clicking `gui\aicb-ui.exe`. No installation and no administrator rights are required.
3. For an MCP client, use `cli\aicb.exe`. It is **not** on `PATH`; only the installer adds it. Give the full path, with doubled backslashes in JSON:

```
{
  "mcpServers": {
    "aicb": {
      "command": "C:\\path\\to\\this\\folder\\cli\\aicb.exe",
      "args": ["mcp"]
    }
  }
}
```

The extracted folder also contains `LIESMICH.txt` (a bilingual readme with the same notes), `EULA.md`, `LICENSE.txt` and `THIRD-PARTY-NOTICES.txt`. To remove the portable copy, delete the folder; your data stays (see below).

Note: An installed copy and an unpacked copy share the data folder `%APPDATA%\AIContextBuilder\` and therefore the same database. If the unpacked copy is newer than the installed one, it migrates the database to its schema, and that migration is one-way: the installed copy will afterwards refuse to start and report that the database was written by a newer version. That is a safeguard, not a defect. Two ways out: bring the installed copy to the same version, or point `Settings > Storage` at a different database file (which starts you with an empty one there).

2.6 One installation per machine

Both the .NET tool and the installer put an `aicb` command on `PATH`, and the two copies can drift apart:

- The installer writes the **machine-wide** `PATH`; the .NET tool installs into your **user** profile.
- Windows builds the `PATH` of a new process from the machine part first, then the user part. With both installed, the installer's copy is therefore the one that runs, for every console and every MCP client.
- `dotnet tool update` would then update a copy that nothing starts, without any error.

Install one of them per machine:

Situation	Install
Windows, and you want the desktop app	the installer only; it contains the same MCP server and CLI, so one update brings app and server to the same version
Everything else (Linux, macOS, CI, or no desktop app)	the .NET tool only

The installer detects an existing .NET tool and offers to remove it (selected by default). It removes it in the account of the signed-in user, not in the administrator account the setup runs as.

For the opposite order, installer first and `dotnet tool install` later, `aicb init` warns when it finds more than one `aicb` on `PATH`, lists which copy runs and which one is ignored, and names the fix: with the desktop app, keep the installer and run `dotnet tool uninstall -g aicb-roslyn-mcp`; without it, uninstall `AI Context Builder` in Windows Settings > Apps.

2.7 The Windows SmartScreen warning

From 0.5.503.1 on, the Windows files are digitally signed, with the publisher `Gregor Dadera` and a timestamp: the installer, the uninstaller, and `gui\aicb-ui.exe`, `cli\aicb.exe` and the product's own `AIContextBuilder*.dll` files inside the ZIP. Versions up to 0.5.502.1 were not signed.

Signing does not stop SmartScreen from warning about a new release. SmartScreen also weighs how much download history a file has, so for a fresh version Windows may still show "Windows protected your PC" until that version has been downloaded often enough. What signing changes is the warning itself: Windows names `Gregor Dadera` as the publisher instead of an unknown publisher. To continue:

1. Choose `More info`.
2. Choose `Run anyway`.

This applies to the installer as well as to `gui\aicb-ui.exe` and `cli\aicb.exe` from the ZIP. The same note is in the `LIESMICH.txt` that ships with the ZIP.

To check the signature yourself, right-click the file, choose `Properties` and open the `Digital Signatures` tab: it lists `Gregor Dadera` as the signer, and `Details` shows whether the signature is valid. One of the files listed above, from version 0.5.503.1 on, without that tab or with another signer is not the published file. The .NET runtime and library files beside them carry their own publishers' signatures.

The NuGet package of the .NET tool is signed by nuget.org, as before; nothing changes for the .NET tool.

2.8 Verifying your download

The release notes for each version list the SHA-256 checksum of every published file. After downloading, compare the file against that value, for example in PowerShell:

```
Get-FileHash .\AIContextBuilder-Setup-<version>.exe -Algorithm SHA256
```

or on Linux and macOS:

```
sha256sum AIContextBuilder-<version>-win-x64.zip
```

2.9 Updating

There is no automatic update: the product does not check for updates, and it does not install anything by itself. How you update depends on the form you installed:

Form	How to update
.NET tool	`dotnet tool update -g aicb-roslyn-mcp`
Installer	download the newer `AIContextBuilder-Setup-<version>.exe` from the release page and run it
Portable ZIP	download the newer ZIP and replace the extracted folder

The installer recognizes an existing installation (it uses the same application identity) and updates it in place; it asks you to close a running instance before it replaces files. If you chose a different install folder in an earlier run, the setup removes the previous `PATH` entry, so `aicb` does not keep resolving to the old copy.

To find out which version you have:

- CLI: `aicb --version`
- Desktop app: `Settings > About` shows the version in the header.

Your data is not part of an update: it lives in `%APPDATA%\AIContextBuilder\` and survives installation, update and uninstall.

Note: A newer version migrates the database schema to its own shape, and that migration is one-way. This matters only if you keep two copies of different versions that share the data folder, for example an installed copy and an unpacked ZIP. In that case, do not start the newer copy last. A copy whose database was migrated by a newer version refuses to start and reports that the database was written by a newer version of AIContextBuilder. Bring the older copy to the same version, or point `Settings > Storage` at a different database file.

2.10 Uninstalling

Form	How to uninstall	What remains
.NET tool	`dotnet tool uninstall -g aicb-roslyn-mcp`	your data in `%APPDATA%\AIContextBuilder\`
Installer	Windows Settings > Apps > `AI Context Builder` > Uninstall	your data; the uninstaller removes the `PATH` entry it added
Portable ZIP	delete the extracted folder	your data

Note: Uninstalling never deletes your data. That is deliberate: the folder holds your database, and the product has no way to restore it. To remove it too, see the next section.

2.11 Removing all data

What the product offers

Only two functions delete recorded data in bulk, and both are in the desktop app:

Function	Where	What it does
`Clear` (dialog `Clear Usage Data`)	`MCP Usage` page	Deletes the recorded MCP tool calls. Two scopes: `Clear everything`, or `Clear calls older than` a date you pick. Before anything is deleted, an export is written; canceling the file picker aborts the whole operation. A second confirmation names how many calls will be deleted and warns that the notes written on those calls are deleted with them and are not part of the export
`Clear API Key`	`Settings > Model Profiles`	Removes the stored API key of the selected model profile from the Windows Credential Manager. Later runs against that profile fail until you enter a new key

The desktop app also offers narrower delete actions for individual sessions, snapshots, MCP-usage notes or imported usage snapshots, and user-created master-data entries.

There is no global "delete all my data" button. No CLI verb deletes anything; the seven verbs are `init`, `analyze`, `export`, `import`, `list`, `mcp` and `call`. The MCP tool `usage_report` is read-only. If you use only the CLI or the MCP server, the only in-product way to remove the usage records is to delete the database file, which also carries every session, snapshot and profile.

The complete removal list

To remove everything AIContextBuilder has written on a machine:

1. `%APPDATA%\AIContextBuilder\` - delete the whole folder. It contains: - `user-data\aicb.acb` - the SQLite database: solutions, sessions, snapshots, profiles, and the MCP usage records; - `app-settings.json` - bootstrap settings (storage paths, recent lists); - `aicb.mcp.json` - optional MCP server configuration, if you created one; - `backups\*.zip` - automatic backups, if they are switched on (on by default for a new installation); - `aicb.log` and `aicb.log.1` to `aicb.log.3` - the desktop app's warning and error log; - `load-perf.log` - solution-load timings, if enabled; - possibly `templates.json` and `node-overrides.json` from earlier versions; the data they held now lives in the database.

2. **Windows Credential Manager** - remove every entry whose name starts with `AIContextBuilder.` (your model-profile API keys, for example `AIContextBuilder.ApiKey:<profile>`). These entries live outside the data folder and are not removed by deleting it.
3. **A storage location you changed yourself** - if you set a different `Base path` or database path in `Settings > Storage`, the database and the backups live there. Note that `app-settings.json` always stays in `%APPDATA%\AIContextBuilder\`, whatever `Base path` says, so a moved storage location means two folders to clean up.
4. **Per analyzed solution:** `<SolutionName>.aicb.json` next to the `.sln`. It is deliberately committed to your repository and shared with everyone working on that solution, so removing it changes the configuration for the whole team. An interrupted write can leave a `<SolutionName>.aicb.json.tmp` beside it.
5. **Per project wired by** `aicb init` **or the** `install_agent_hooks` **tool** - these entries are merged into files that belong to you, so edit them instead of deleting the files: - the `aicb` entry in the project's `.mcp.json`; - the skill files under `.claude/skills/` (`aicb-csharp-context`, and with `--skills=all` also `aicb-code-review`, `aicb-code-simplifier` and `aicb-usage-check`); - Claude Code: `.claude/hooks/aicb-symbol-guard.mjs` and the aicb entry in `.claude/settings.json`; - Codex: `.codex/hooks/aicb-symbol-guard.mjs` and the aicb entry in `.codex/hooks.json`; - OpenCode: `.opencode/plugins/aicb-symbol-guard.ts`, `.opencode/plugins/guard-wiring.mjs`, `.opencode/plugins/aicb-symbol-guard.mjs` and the aicb entry in `opencode.json`.
6. **Files you exported yourself** - context documents written with `aicb analyze`, `aicb export` or `export_markdown`, usage reports as JSON, session exports, run templates and constellations.
7. `%TEMP%\acb-*` - temporary working folders. They are removed when the command that created them ends; after a crash one may remain.

Note: On Linux and macOS the `%APPDATA%` token resolves to the platform's application-data folder rather than to a Windows path.

What the logs contain

- `aicb.log` - warnings and errors from the desktop app only, with timestamp, level, category and message. It rotates at 1 MiB and keeps three archives (`aicb.log.1` to `aicb.log.3`); it can be copied while the app is running. The CLI and the MCP server do not write it.
- `load-perf.log` - solution-load phase timings from the desktop app only: phase names, milliseconds and a context label. It does not rotate and grows with every solution load. You can switch it off in `Settings > General` with `Record solution-load performance timings`.
- The CLI and the MCP server write no log file; their diagnostics go to standard error.

Note: The MCP server records one row per tool call in the local database: timestamp, tool name, success, duration, client name and version, and a truncated error text. Nothing is transmitted; the rows stay on your machine and are what the `MCP Usage` page shows and its `Clear` button deletes.

2.12 Support and security reports

Questions, bug reports and feature requests go to GitHub Issues at https://github.com/gregordadera/aicb-roslyn-mcp/issues. Please include the version (`aicb --version`, or `Settings > About` in the desktop app) and, for MCP problems, the output of the `server_info` tool.

Report security issues privately, not as a public issue: by email to **aicb@dadera.de**, or through GitHub's `Report a vulnerability` on the repository's *Security* tab. You will get an acknowledgment, and a fix or mitigation will be coordinated before any public disclosure. There is no supported-versions table and no response-time commitment.

Note: Opening a solution runs its MSBuild build logic, the same thing that happens when you open it in Visual Studio or run `dotnet build`. Only analyze solutions you trust. The full threat model is in `SECURITY.md` in the public repository.

2.13 Building from source

AIContextBuilder is closed source, and the public repository contains the documentation, the license and the releases, not the source code. Building from source is therefore not a way to install the product. Use the .NET tool, the Windows installer or the portable ZIP described above.

3 Core concepts

This chapter defines the vocabulary the rest of the manual uses. The same words appear in the graphical interface, on the command line and in the answers of the MCP server, so a term you read here is a term you can search for on screen or in a tool answer. Read this chapter first - every other chapter builds on it.

3.1 Solution

A **solution** is a Visual Studio solution: the `.sln` file plus all the projects it contains. It is the unit the tool analyzes - without a solution, nothing happens.

You meet the solution in several places:

- In the GUI, in the `Workspace` area (the sidebar entry of the same name). Its left list is headed `Solutions`; each entry is one solution the application knows about.
- In the `Solution Tree` of the Context Builder, which shows the loaded solution's projects and files.
- On the command line: `aicb analyze --solution <path-to-.sln>`, and `aicb call <tool> --sln <path-to-.sln>` for a single tool call.
- In the MCP server: a `sessionId` argument accepts the absolute path of a solution file (`.sln`, `.slnx` or `.slnf`) directly and analyzes it on first use.

Note: the application also stores an entry per solution that carries the profiles assigned to it, its sessions and its snapshots. "The solution" in this manual always means the analyzed project structure, not that stored entry.

3.2 Project

A **project** is a `.csproj` inside the solution. An analysis walks every project of the solution.

A project that targets several frameworks (multi-targeting) appears **once per target framework** in the analysis. Several tools present a deduplicated view instead and keep only the preferred (normally newest) target-framework instance. This view is not lossless: a symbol or fan-in edge that exists only in an older target-framework instance is not reported by those deduplicated tools.

You meet projects as the second level of the `Solution Tree`, in the `PROJECT` section of a context document, and in the `project` field of tool answers.

3.3 Namespace, type and member

Namespace - the C# namespace a type is declared in. It is part of a type's full name, and it is the axis by which you filter and scope: `Exclude Namespaces` in the settings, a namespace prefix as a tool `scope`, or a namespace pattern in a layer profile.

Type - a class, a struct, an interface, a record or an enum.

Member - a constituent of a type: a method, a property, a field, an event or a constructor.

The `Solution Tree` shows the physical side of this: the solution, its projects, their folders and their files. Under each C# file the analysis adds the file's types, and under each type its methods, so you can select at any level. The checkbox `Public methods only` hides non-public methods in the tree.

3.4 Unit

A **unit** is everything that has an executable body and can therefore be analyzed. That is more than a declared method:

- **Methods** - ordinary method declarations.
- **Accessor units** - every property, indexer or event accessor with a body (`get { … }`, `set => ...`, `init`, `add`, `remove`), plus the getter of an expression-bodied property such as `public int Total => _a + _b;`. An auto-property accessor (`get;` / `set;`) has no body and produces no unit.
- **Constructor units** - one per constructor that executes something: a declared constructor is analyzed as a whole declaration, so calls inside `: base(...)` / `: this(...)` arguments are captured, together with the member initializers it runs. For a constructor that is declared elsewhere - the implicit parameterless constructor, a C# 12 primary constructor, a constructor in another partial file, or the implicit static constructor - only the initializers are analyzed. A struct's default constructor runs no initializer, and a record's copy constructor copies instead.
- **Top-level statements**, **Razor components** and `.cshtml` **views** - analyzed by their own passes. Their calls and type references are recorded with a `... (top-level)` or `... (razor)` source label (a view as `<path>.cshtml (razor)`), so they appear as callers in fan-in answers.

The distinction is not cosmetic: a private method called only from a property setter would have no caller at all if setters were not units, and the dead-code and fan-in answers would be wrong on live code.

You meet units as the counters of tool answers (`methodsScanned` counts the units a scan looked at) and as addressable names: accessor units are named by their accessor (`get_Items`, `set_Items`), constructor units by `.ctor` (`.cctor` for the static constructor).

Note: a partial type is analyzed fragment by fragment. An initializer in one fragment and the constructor that runs it in another yield two units under the same key rather than one merged row.

3.5 Analysis

An **analysis** reads the solution's source files with the C# compiler's semantic model (Roslyn) and produces the model this chapter describes: the tree of projects, files, types and members, together with the measured facts and the resolved semantic values.

An analysis runs when you open a solution in the GUI, when you call `aicb analyze`, and when an MCP tool initializes itself from a solution path or refreshes its session.

- Which projects count as test projects is configurable per solution (`Test Profile`); production-focused tools exclude them by default.
- A run that cannot resolve every project's references records that its resolution was incomplete, and the affected tools say so in their answers instead of presenting a short count as a measurement. If you analyze a freshly cloned repository, build it once (or restore its packages) and analyze again to get a complete graph.

3.6 The tree

An analysis produces a tree. Each level narrows the one above it:

```
Solution
└─ Project          (one instance per target framework)
   └─ File          (name, path, namespace)
      └─ Type       (class, struct, interface, record, enum)
         ├─ Method
         ├─ Property
         ├─ Field
         ├─ Event
         └─ Constructor
```

Parameters belong to methods and constructors.

The tree is the backbone of every answer: a scope narrows it (a project, a namespace prefix, a file), and the sections of a context document follow its levels.

3.7 Facts

A **fact** is a measured property of a symbol, derived deterministically from the source code. Facts are the hard evidence; the semantic values described in the next section are resolved on top of them.

Fact	What it records
Cyclomatic complexity	McCabe complexity. `0` = not computed because no symbol could be resolved (or from an older snapshot), `1` = linear, higher = more paths. Abstract and extern members without bodies are measured as `1`.
Cognitive complexity	How hard the body is to understand (nesting, `else` / `catch`, runs of boolean operators). `0` = linear or without a body and is a valid score, not a sentinel.
Lines of code	The lines of the declaration that carry at least one token - signature, attributes and body. Blank lines and comment-only lines are not counted; a line with code and a trailing comment is; a token that spans several lines counts each of them. This is deliberately not the raw line span, because that number grows with every comment and the best-documented method would read as the longest one. `0` = not computed.
Side effects	The outside-world contact of the body; see below.
Called project methods	The solution methods the body invokes. A creation (`new X(...)`) counts as a call to the solution constructor it binds.
Used types	The types the body names.
Injected dependencies used	Which constructor-injected dependencies the body actually uses.
Created types	The types the body instantiates with `new` (including target-typed `new`).
External calls and reads	Which BCL and third-party members the body calls, and which external state it reads (for example `System.DateTime.UtcNow`).
Leaked disposables	Disposables created with `new` whose ownership is neither transferred nor disposed - a resource-leak smell.

Every unit is registered under a stable, fully-qualified lookup key built from namespace, containing type, method name and parameter types. Edges are registered under that key, so overloads, namespaces and generic substitutions stay apart; answers that return callers or callees use these keys.

Side effects

The vocabulary of side effects is closed and has eight values:

Token	Meaning
`io`	Filesystem and stream access. A `System.IO` type is judged by contact, not by topic: `Path` and `MemoryStream` carry no effect, `File` and `FileStream` do.
`network`	HTTP, socket, mail and cloud-storage calls. Addresses and message surfaces (`IPAddress` , HTTP message types) carry no effect; `Dns` , `Sockets` , `HttpClient` and comparable clients do.
`database`	Database drivers and ORMs (for example ADO.NET, Entity Framework Core, Dapper, Npgsql, MongoDB, Cosmos).
`serialization`	(De)serializer calls (for example `System.Text.Json` , Newtonsoft.Json, `XmlSerializer` , MessagePack, protobuf, YamlDotNet).
`logging`	Logger calls (for example `Microsoft.Extensions.Logging` , Serilog, NLog, log4net).
`cache`	Calls into an external cache API (for example `Microsoft.Extensions.Caching` , StackExchange.Redis, 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, Kafka, NATS, MediatR, 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.

Effects include the transitive ones, along calls and creations: a method that calls a project method which hits the database is a database method itself, and so is a method that builds an object whose constructor hits the database. The classification judges by **contact, not by topic** - `Path` and `MemoryStream` carry no `io` effect, `File` and `FileStream` do.

You meet the resolved side-effect classification in `find_by_side_effects` (which can also list only pure methods with `purity: pure` or only impure ones with `purity: impure`) and in the quality-profile option `Side-effect concentration (methods mixing 3+ effect categories)` . A context document does not render that resolved classification in `SEMANTICS` : an explicit `<ai>` annotation can appear as `SideEffects` in `AI_TAGS` , and a provenance line can state `verified-absent: sideEffects` .

3.8 Semantic axes and provenance

Semantic axes are derived values on a type or a method - what a symbol is for, where it belongs, how it behaves. Unlike facts they are not measured; they are *resolved*, and every resolved value carries its origin.

A type carries up to 16 axes, a method up to 8:

Axis	Applies to
`Context`, `Domain`, `Role`, `Layer`, `Priority`, `Complexity`, `SideEffects`, `Stability`	types and methods
`Responsibility`, `Pattern`, `DependencyType`, `Determinism`, `DataAccess`, `Interaction`, `Validation`, `ErrorHandling`	types only

Each axis is resolved through a fixed priority chain:

1. **Fact** - a deterministic value derived from the source. A fact is never overridden by an annotation.
2. **Explicit absence** - the developer declared the axis as none (see the `<ai>` annotation below). The value stays empty and is reported as verified absent.
3. **Explicit value** - a value the developer asserted in an `<ai>` annotation.
4. **Inference** - a heuristic guess, used only when none of the above applies.

Provenance is written into the output, so a reader can tell a measured value from an assertion and from a guess. In a full `<SEMANTICS>` block the last line is a `Source:` line:

Form	Meaning
`Source: Resolved`	No per-field provenance was recorded; the marker stands alone.
`Source: Inferred`, `Source: Ai`, `Source: Fact`	Every emitted field has the same origin (short form).
`Source: Resolved (role=ai, layer=inferred)`	A mixed block: one `field=token` pair per field.
`Source: Resolved (role=inferred; verified-absent: context)`	Fields the developer declared absent are listed after `verified-absent:`.

The tokens are `fact`, `ai`, `inferred` and `verified-absent`. In a compact `<SEMANTICS compact>` block the same information appears as an optional `src:` line, for example `src: role=ai, layer=inferred`; a compact block without a `src:` line is fully inferred.

The context document states the rule for reading these values itself: treat `inferred` values as hints, not ground truth, and do not rely on them where correctness matters.

The same vocabulary is available to queries: `confidence_map` returns the provenance of each semantic field, `coverage_gaps` lists axes that are verified absent, and `assert_absence` checks such a declaration.

3.9 The `<ai>` annotation

A developer can override the heuristic for a type or a method by writing an `<ai>` block into its XML documentation comment (`///`). The analysis reads it and uses it as the explicit channel of the resolution above.

Two syntaxes are accepted and may be mixed freely: colon-separated (`role: Repository`) and XML-attribute (`role="Repository"`). Several attributes may share one line.

```
/// <ai
///   role="dto"
///   responsibility="Carries the inputs of a single code-analysis run from the application layer to
CodeAnalyzer."
///   layer="Application"
///   stability="Evolving"
/// />
```

Three shapes are recognized: the element form `<ai> … </ai>`, a self-closing block whose attributes run over several lines and end with `/>`, and a single-line `<ai role="..." />`. Inside an element block, a line that merely ends in `/>` is ordinary content.

- Where it can stand: on a method declaration, and on the declaration of a class, struct, interface, record or enum.
- Keys: the axis names listed above (`role`, `layer`, `domain`, `context`, `priority`, `complexity`, `sideEffects`, `stability`, and for types also `responsibility`, `pattern`, `dependencyType`, `determinism`, `dataAccess`, `interaction`, `validation`, `errorHandling`). Keys are matched case-insensitively; an unknown key is ignored.
- Sentinel values: `none`, `null` or `empty` (any capitalization) suppress the inference for that field. The value is never written out as text - the field stays empty and is reported as `verified-absent`.
- `importance` is accepted as a backward-compatible alias for `priority`. The legacy values `important`, `less important` and `nice to have` resolve to `high`, `medium` and `low`.

A second example, in colon syntax, with one axis deliberately switched off:

```
/// <ai>
/// role: Repository
/// stability: Stable
/// sideEffects: none
/// </ai>
```

The result for such a type: `role` and `stability` appear as `ai` in the provenance, `sideEffects` is reported as `verified-absent` and is not inferred.

3.10 Edges

The analysis records the relationships between symbols as **edges**. These are the kinds you will meet:

Edge	Direction	Where you meet it
calls	method → method	`call_graph`, `METHOD_GRAPH`
creates	method -> the solution constructor a `new X(...)` binds	`METHOD_GRAPH`, a method's `Calls:` list, `impact_of_change`, `find_tests_for`, `coverage_gaps`
called by (fan-in)	method → callers	`find_usages`, `impact_of_change`, `METHOD_USED_BY_GRAPH`
operator use	unit -> user-defined operator or conversion (`a == b`, a cast, an implicit conversion)	`find_usages`, `impact_of_change`, `find_tests_for`

Edge	Direction	Where you meet it
depends on	type → type	`CLASS_DEPENDENCY_GRAPH`, the efferent-coupling metric (`Ce`)
implements	type → interface	`find_implementations`, `INTERFACE_RELATIONS`
inherits / overrides	type → base type	`get_type_hierarchy`, `find_overrides`
markup references type	view → type	`find_usages`, `find_dead_code` (`kinds: type`)
markup binds member	view → member	`find_binding_usages`
member access	unit → property, field, event or enum member	`find_usages`
markup calls member	view → method	`find_usages` (for example a Blazor event handler referenced only from markup)
top-level statements reference	`Program.cs` → type (every type the statements name: `typeof`, type arguments, casts, patterns, declared locals, the types in a called method's signature)	`find_usages`, `impact_of_change`, `find_dead_code` (`kinds: type`)
top-level statements create	`Program.cs` -> the type a `new X(...)` there creates	`instantiation_sites`, `lifecycle_of`, the constructor form of `find_usages` (`Type.Type`)
assembly attribute	the file that applies `[assembly: ...]` or `[module: ...]` -> attribute type, labeled with the file path and `(assembly)` or `(module)`, for example `src/App/Properties/AssemblyInfo.cs (assembly)`	`find_usages`, `impact_of_change`, `find_dead_code` (`kinds: type`)

"Markup" covers WPF `.xaml`, Avalonia `.axaml` and Blazor `.razor` files. In a `.razor` file this includes the C# in `@code` blocks and code-block statements, component-parameter values whose parameter is typed and not a `string`, and calls through an injected member (`@inject`, `[Inject]`) or through a field or property of the component itself, which count for the member on that member's type.

An MVC or Razor Pages view (`.cshtml`) counts where the Razor source generator compiled it in the analysis: what the view calls, reads or names - an `@attribute [X]` included - is a use, credited to the view file. A type injected with `@inject` in a `_ViewImports.cshtml` is credited to that file once, not to every view below it. A view the generator did not compile - a view compiled only at run time, or one the generator did not run for - is not read; a zero fan-in answer and a dead-code candidate list then name those views (`VIEWS NOT READ: ...`), so a missing user there is not read as "unused".

An attribute is a type reference wherever it is applied: on a type, a member, a return value, a parameter, a type parameter, or a local function or lambda.

The type dependency list is the union of the injection dependencies and the usage dependencies of a type, deduplicated, sorted and with self-references removed. Which member types it names is described under `DEPENDENCIES` in the chapter on the context document.

Note: the type dependency list is built from short type names, so two types with the same simple name in different namespaces are not distinguished there; where a collision occurs, the first one in the stable solution order wins. The circular-dependency check works on fully-qualified names and is not affected.

3.11 Fan-in and impact

Fan-in is the set of places that name a symbol - the answer to "who calls this?" It is the opposite direction of the call graph.

Impact is the transitive closure of the fan-in: everything that would be affected indirectly by a change to the symbol, including consumers that a compiler does not check.

`find_usages` lists the direct mentions, one level, each with the file and the line its declaration starts on. `impact_of_change` returns the closure together with a `risk` level. The risk is calibrated on the production fan-in; callers in test projects are excluded, because exercising a symbol from tests is its designed fan-in and updating those tests is mechanical. On an interface or abstract member or type, the production types that implement it count toward the risk as well, because a signature change breaks them too.

Users that share a name - two types called `Startup` in different projects, for example - are listed by `find_usages` as separate items, one per declaration, and the items that would otherwise read alike add `fullName` (for a type) and `project`. A type of the same simple name but another arity or namespace that derives from, holds or creates the queried type is one of its users (a generic `Identity<T1, T2> : Identity`, for example); a reference that names only the type itself is not. `impact_of_change` walks its closure over declarations in the same way, so `transitiveCount` counts one entry per declaration.

User-defined operators and conversions have a fan-in of their own: `a == b`, a cast and an implicit conversion are uses of the declared operator. The fan-in tools take an operator by its metadata name, such as `op_Equality` or the qualified `Money.op_Implicit`.

Note: the `risk` level measures the fan-in, not the semantics of your concrete change. A `high` on a purely additive change is expected and is not a stop signal.

3.12 Dead code and cycles

Dead code is a symbol that nothing in the analyzed scope names. The scope is what matters: reflection, dependency injection resolved from strings and callers outside the solution are invisible to a static analysis, so a missing fan-in is a place to look, not a proof. The tools report their candidates with that caveat and name what they cannot see.

Cycle - a chain of types or namespaces that leads back to itself. The compiler forbids cycles between projects, but it allows cycles between namespaces; `detect_circular_dependencies` reports groups of namespaces that depend on each other and shows an example edge for each link so you can see why the cycle exists.

3.13 Layers

A **layer** is an architecture layer such as `Domain`, `Application` or `Infrastructure`. A **Layer Profile** is a named rule set that maps types to layers, and it also defines how strictly a violation is judged (`Advisory` or `Strict`).

You meet layers in the `Layer Profiles` page of the settings, in the `Layer Profile` picker per solution, in the `--layer-profile` option, in the `.aicb.json` sidecar file and in the `LAYER_MAP` section of a context document. The `Layer` semantic axis on a type or a method is the resolved layer of that symbol.

3.14 The context document

The **context document** is the product of an export: a Markdown document written for a language model to read, in the **AI-Builder-MD** format. It contains the selected part of the analysis - projects, files, types and methods with their facts and semantic values - organized in named sections such as `SPEC`, `META`, `DOMAIN`, `LAYER_MAP`, `CLASS`, `INTERFACE`, `ENUM` or `QUALITY_FINDINGS`. A `<METHOD>` is a block inside a type section (and becomes an entry under `methods:` in YAML), not a top-level section. A `Source:` line inside a `<SEMANTICS>` block tells the reader where each semantic value came from, and `COMPRESSION_LEGEND` and `PATH_LEGEND` explain the notation of the document itself.

The same content can be written in two notations:

- `Tag` - the established tag markdown, with elements such as `<CLASS>...</CLASS>`.
- `Yaml` - the same information as idiomatic YAML, a lossless re-notation rather than a different selection of content.

`Yaml` is the default. You can change the notation per document: `Document Notation` in the export panel of the GUI, `--format tag|yaml` on the command line, or the `format` parameter of the MCP tool `export_markdown`. The slice tools `get_context`, `explain_symbol`, `pack_for_task` and `prepare_task` do not expose a notation switch. The file always keeps its `.md` name.

You produce a context document with the export in the GUI, with `aicb analyze --output`, with `aicb export`, and with the render tools of the MCP server - `export_markdown` for the whole solution, or a bounded slice such as `get_context`, `explain_symbol` or `pack_for_task` for one symbol's neighborhood.

3.15 Sessions and snapshots

Sessions

A **session** is a named, saved working state of a solution: the selection in the `Solution Tree`, the per-node detail-level overrides, the prompt text and the frozen template. You open it later and continue where you left off. Sessions live in the `Sessions` sub-tab of the `Workspace` area; `Save Session` creates one.

Note: this is not the **MCP session** - the analysis result an MCP server keeps in its process, addressed by a `session_id`. It has no name and no database row, and it is gone when the process ends. Where both could be meant, this manual writes `MCP session`; see "Sessions and staleness".

Snapshots

A **snapshot** is a frozen analysis result of a solution at a point in time. A run is tied to a snapshot, so a later re-analysis does not act back on completed runs. Snapshots live in the `Snapshots` sub-tab of the `Workspace` area:

- `Create Manual Snapshot` creates a named snapshot you keep.
- `Diff Two Snapshots` compares two snapshots.
- `Max snapshots per solution` limits how many automatic snapshots are kept. One automatic snapshot is written on every fresh analysis; older ones are deleted when the limit is exceeded. Snapshots you created yourself are never deleted and do not count towards the limit.

A snapshot stores the analysis as it was when it was written, including the facts of that moment.

Note: the `Snapshots` sub-tab of the `MCP Usage` page shows imported usage reports, not analysis snapshots. Imported reports sit beside the live data as named sets and are never merged into it.

3.16 Insights and severity

An **Insight** is a finding of the built-in code-quality analysis: a place where the tool found something worth mentioning, with a severity, a producer group and locations. You meet insights in the `Insights` tab - its header shows `Insights (N)` while findings are unseen - in the `list_insights` and `get_insight` tools, and in the `QUALITY_FINDINGS` section of a context document.

Note: `Findings` names something else. It is the region of the `Reasoning` panel that shows findings parsed from the language model's answer, not the results of the product's own analysis.

Severity has four levels: `Info`, `Ok`, `Warning` and `Critical`. The ordinal value is not a weight; for sorting and scoring the product uses `Critical` = 10, `Warning` = 3, `Info` = 0.5 and `Ok` = 0. On the wire the levels are lowercase tokens: `info`, `ok`, `warning`, `critical`.

False positive - a finding that is reported but is not real. The product carries its own taxonomy of them and does not claim that none occur: the analysis tools state their blind spots (see the fan-in, dead-code and markup notes above) rather than presenting every result as complete.

4 How the analysis works

AICB does not answer from a text index. It opens your solution the way an IDE does, builds a semantic model of the code with the C# compiler platform (Roslyn), and derives every fact it reports - callers, side effects, layers, tests, metrics - from that model. The same analysis feeds all three surfaces: the desktop app, the `analyze` CLI verb and the MCP server.

This chapter describes what happens between opening a solution and the first answer: what the analysis takes in, how a solution is loaded, which semantic layers are computed on top of the model, and where the accuracy limits are.

4.1 What goes in

The analysis takes an absolute path to a solution file. Three formats are accepted:

Format	What it is
`.sln`	Classic Visual Studio solution
`.slnx`	The XML-based solution format
`.slnf`	A solution filter - a `.sln` plus a subset of projects. Useful when a full solution is too large or too slow to analyze

Two further inputs are read from the solution's own configuration, not from the call:

- **The per-solution configuration** (`.aicb.json` , next to the `.sln`). It can carry layer rules, namespace exclusions, a test profile, the multi-targeting scope (`analyzePreferredTfmOnly`) and the platform of the design-time build (`designTimeBuildPlatform`). Headless CLI/MCP analysis uses its layer and exclusion axes as database-free fallbacks and reads the scope and the platform directly; the sidecar test profile is configuration/setup material rather than part of the headless analysis resolution chain. In the desktop app, layer, exclusion and test content must first be restored into the database through auto-initialization or `Initialize Now` , while the scope and platform keys are read directly. The file is plain JSON and git-tracked, so it travels with the repository.
- `<ai ...>` **annotations** in the source. Authors can annotate a type with fields such as `layer` , `role` , `priority` or `responsibility` , and a member with `layer` , `role` or `priority` ; `responsibility` is type-only. These values are treated as explicit author statements and outrank every heuristic. Sentinel values `none` , `null` and `empty` (any casing) suppress inference for a field and are never emitted as text; the old key `importance` remains a backward-compatible alias for `priority` .

The analysis is a **read-only** operation with respect to your source. It does not edit files, and it does not run your solution's third-party Roslyn analyzers (`get_diagnostics` can run their DiagnosticSuppressors on request, see "The one caveat: MSBuild"). Source generators are different: when AICB builds a project's compilation, Roslyn runs the source generators the project references, inside the AICB process, as an IDE does. AICB does not index their output, with one exception: the code the Razor generator compiles from `.cshtml` views is read, so that what a view uses counts toward fan-in (see "Edges"). Where it reports source-generated members (for example `[ObservableProperty]` or `[RelayCommand]`), it re-derives their names from the attributes in the parsed syntax. A generator whose assembly is missing, for example an in-solution generator project that was never built, cannot run, and its generated code is then missing from the compilation. When that happens, the fan-in answers about a symbol in that project or in a project that depends on it (`find_usages` , `impact_of_change` , `instantiation_sites` , `find_tests_for` , `find_implementations` , `get_type_hierarchy`) carry a `generatedCodeNote` naming the generator

and the repair, even when the list is not empty, and `find_dead_code` marks a candidate in such a project with `missingGeneratedCode: true` - treat it as unjudged, not as dead. The MCP server and the CLI have no outbound network capability at all.

Generated C# that is committed to the repository - a `*.Designer.cs` or `*.g.cs` file outside build output - is read like any other source file: its references count toward fan-in, but its code is not judged by the quality rules, the dead-code tools or `coverage_gaps`. The same file names under the project's `bin` or `obj` folder are build output and stay out. A `bin`, `obj` or `.vs` folder that contains the project itself, such as a checkout under `~/bin/repo`, is not build output: such a checkout is analyzed normally.

> **Security note - opening a solution runs its MSBuild logic and its source generators.** To resolve references and determine what to compile, AICB performs MSBuild's *design-time build*. That evaluates and runs the project's own MSBuild logic: its `.csproj`, `Directory.Build.props` / `.targets`, SDK and NuGet imports, and any custom targets or inline tasks. Building the compilation then runs the source generators the projects reference. A deliberately malicious project could therefore run code at solution-open time. This is not specific to AICB - it is inherent to every MSBuild-based tool, including Visual Studio and `dotnet build`. **Treat pointing AICB at a solution as equivalent to building it: only analyze solutions you trust.**

4.2 Opening a solution

MSBuild registration

Before the first project can be loaded, the MSBuild toolchain must be located and registered. This happens exactly once per process; a registration that another host (for example Visual Studio) has already made is honored rather than repeated.

AICB searches in this order:

1. The default MSBuild discovery of the locator.
2. The newest Visual Studio instance known to the locator (Windows).
3. `vswhere.exe` under `%ProgramFiles(x86)%\Microsoft Visual Studio\Installer` (Windows) - this catches Visual Studio versions the locator does not know yet.
4. The newest .NET SDK under `%ProgramFiles%\dotnet\sdk` that contains an `MSBuild.dll` (Windows).

On Linux and macOS only the default discovery applies; a .NET SDK must be installed. If nothing is found, the message states what was tried and names the possible causes: Visual Studio without the '.NET desktop development' or 'MSBuild Tools' workload, or neither Visual Studio nor a .NET SDK installed. A `global.json` next to the solution can pin which SDK version is used when several are installed.

The solution registry

Loaded solutions are kept in a reference-counted cache keyed by the absolute solution path (case-insensitive). Two properties matter in practice:

- **Concurrent access to the same solution coalesces into a single load.** If two tabs, two agents or a tool call and a GUI action open the same solution at the same time, only one load runs; the others wait for it. A warm hit on a *different* solution does not wait behind a slow load - the long operations run under a lock per solution path, while the short bookkeeping is separate.

- **A loaded solution stays warm while at least one session uses it.** When the last user releases it, the workspace is disposed and the memory is freed.

Waiting for a load in progress is observable. The desktop app and the CLI wait indefinitely by default. The MCP server bounds the wait to 45 seconds and then answers with a "load in progress" message that names the solution and the running operation, instead of blocking past the client's own timeout. The bound can be changed with the environment variable `AICB_MCP_LOAD_WAIT_MS` (milliseconds). If a load is stuck, the message recommends waiting for it to finish or analyzing a smaller solution or a filter (`.slnf`).

The design-time build cache

The expensive part of opening a solution is the MSBuild design-time build. Its result is cached per solution, so a later process can skip MSBuild when the project shape has not changed.

Setting	Value
Default location	`%LOCALAPPDATA%\AIContextBuilder\design-time-build-cache`
Relocate	Set `AICB_DESIGN_TIME_BUILD_CACHE` to a directory path
Disable	Set `AICB_DESIGN_TIME_BUILD_CACHE` to `0`, `off` or `false`
What invalidates it	Changes to project-shaping files: project files; any `*.props` / `*.targets` (including `Directory.Build.props`, `Directory.Build.targets` and `Directory.Packages.props`); `*.sln`, `*.slnx` and `*.slnf`; `global.json`; `nuget.config`; `packages.lock.json`; `.editorconfig` / `.globalconfig`; `*.csproj.user`; `obj/project.assets.json`; and the installed SDK versions. Source edits do **not** invalidate it - a rebuild is only about references and compilation structure, and the source is re-read anyway. Adding or removing a source file or a view (`.cs`, `.xaml`, `.cshtml`, `.razor`) below the solution folder does, because it changes the set of documents a project compiles
Cleanup	A cache file that is neither written nor hit for 30 days is deleted the next time a store finds more than 32 cache files. A hit touches the file, so a solution you open regularly never ages out

Each cache file is specific to one solution (a hash of the canonical path); two solutions never share one. If the cache is unusable for any reason, AICB silently falls back to running MSBuild - the cache is an optimization, never a requirement.

Two solution shapes are cached like any other, so their second open skips MSBuild as well:

- A project whose build lists a compile item that has no file on disk, typically a file a pre-build step or a code generator writes. `get_diagnostics` names such items under `missingCompileItems` when the project has errors in the answer (see "The solution was never built").
- A `.sln` with Visual Studio Web Site entries (a URL or a folder path, no project file). In most cases such an entry is not counted as a failed design-time build and does not appear in `get_diagnostics` `designTimeBuildFailures`; it still does when a loaded project lost a project reference. A missing `.csproj` is still a failure, and Web Site entries in a `.slnx` are not recognized.

The build platform

A solution that defines an `Any CPU` platform is design-time built as before, with no platform passed. A solution without `Any CPU` (only `x64` and `ARM64`, for example) is design-time built at the platform its default solution configuration builds its projects at, when the solution leaves no doubt about that platform; where it does leave

doubt, the solution loads without a platform, as before 0.5.503.1. A load at the wrong platform shows up as unresolved projects, missing generated code and build failures about the platform or the architecture.

To choose the platform yourself, set `designTimeBuildPlatform` in the solution's `.aicb.json` (see "The `.aicb.json` sidecar"). `get_diagnostics` reports the decision in its `designTimeBuildPlatform` field, which is omitted when no platform key is set and the solution leaves nothing to decide (it defines `Any CPU`, builds every C# project at AnyCPU, or cannot be read): the platform used, the solution configuration and platforms it read, the projects the solution builds at another platform, and a `note`; a platform taken from the sidecar adds `source: "sidecar"`, and a refused value appears as `refusedSidecarPlatform`.

4.3 The analysis run

One analysis run has a single entry point and takes the following inputs:

Input	Meaning	Default
Solution handle	The loaded workspace. Required	-
Layer profile	The namespace-to-layer mapping to use. When unset, only the built-in role heuristic applies	none
Excluded namespaces	The namespaces to keep out of the type-reference facts (parameters, return types, fields, properties, dependencies). Supplied from the solution configuration when one exists	none
`AnalyzePreferredTfmOnly`	Analyze only the preferred target-framework instance of a multi-targeted project (see "Multi-targeting")	`false`

The run walks every C# project and, per document, builds a syntax tree and a semantic model. A document whose syntax tree or semantic model cannot be obtained is skipped. The analyzable type nodes are `class`, `struct`, `interface`, `record` and `enum`.

Beyond declared methods, the analysis treats accessors and constructors as additional **units** with a body:

- **Accessors** - every property, indexer and event accessor with a body (`get { … }`, `set => ...`, `init`, `add` / `remove`) plus the getter of an expression-bodied property. An auto-property accessor (`get;`) has no body and produces no unit.
- **Constructors** - each constructor that executes something, including member initializers; a constructor declared elsewhere contributes its initializers to the constructor that runs them.

Top-level statements, Razor components and compiled `.cshtml` views are covered by dedicated analyzers that record their call and type references in the fan-in indexes. They are not method units and do not contribute to coverage or dead-code totals. Top-level statements are counted in two places: an unscoped `calls_external` or `find_by_side_effects` query treats each file of top-level statements as one unit, listed as `<path>/Program.cs (top-level)` and counted in `methodsScanned`, with the effects of the project methods and constructors it calls inherited (a namespace-scoped query and `purity: pure` stay methods-only); and `lifecycle_of` counts each top-level file whose creations it read in its `methodsScanned`.

While the run proceeds, progress is reported per document in roughly 1 % steps - at most about 100 updates per run, so the reporting itself stays cheap. A run can be cancelled; the cancellation is passed through the project loop, the document analysis and the Roslyn calls.

After the document phase, a single-threaded post-processing phase consolidates the run's indexes:

1. Resolve XAML `{Binding}` facts against the built types and fold them into the member fan-in.
2. Resolve Razor member references against the component types.
3. Resolve parameters a Razor component hands to child components.
4. Resolve XAML `{x:Type}` references into the type fan-in.
5. Resolve top-level-statement type references into the type fan-in.
6. Apply the state-driven indexes: caller lists, type fan-in, implemented-by lists and resolved semantics.
7. Propagate side effects transitively along the call graph (see "Side-effect classification").
8. Inherit each type's layer onto its methods.

Two results are computed only for a live analysis and are not re-derived from a stored snapshot: the **unresolved XAML bindings** and the **completeness record** described next.

4.4 Incomplete projects

A project is called **incomplete** (unresolved) when its compilation cannot bind its core references - concretely, when `System.Object` is an error type. Typical causes are an unrestored project (no `restore` has run) or a targeting pack the machine does not have.

An incomplete project is one of the most important states to understand, because it decides whether you can trust an answer. Such a project still yields syntax, so the analysis walks it and still produces facts - but the **semantic edges are missing**: an unresolved call site records no caller. Every fan-in answer that follows is short by an unknown number of ordinary references, and nothing about it looks wrong.

The run records its own completeness and reports one of three states:

State	Condition	Meaning
All resolved	At least one project scanned, none unresolved	Everything bound
Incomplete	At least one project scanned, at least one unresolved	The alarm state
Indeterminate	No project scanned	Unknown - no statement is possible

The unresolved projects are named, not merely counted, so that "restore `MyApp.Infrastructure`" is an actionable next step.

> **Note:** "All resolved" does not mean "the project is restored". The check looks at `System.Object`, which a project gets from the base framework rather than from its own restore. A project without a `project.assets.json` typically loads, binds `System.Object` and counts as resolved, while every fact typed by its missing references is gone. What the flag really separates is a *wholly* unbuilt working tree from everything else. `get_diagnostics` lists such a project under `unrestoredProjects` when it has unresolved-name errors (`CS0246`, `CS0234`) in the answer, with a note naming the repair: `dotnet restore`, and on `NETSDK1147` `dotnet workload restore`.

The completeness record travels with the session and is stored with a remembered codebase; a missing value means "unknown", never "fine".

When a run is incomplete, tool answers say so. Answers about fan-in (for example `find_usages`, `impact_of_change`, `find_dead_code`, `instantiation_sites`) carry a leading note that their numbers are short by an unknown number of ordinary references. Answers about side effects are sharper still, because effects are derived by propagating along

call edges: an unresolved call makes a method read as pure. The note in that case tells you to treat the result as a floor and never as a passed purity or architecture check. The same applies to external-call searches and to code-trait queries such as reflection usage.

Practical consequence: **restore and build the solution before you rely on effect, leak or external-call statements.** Fan-in statements survive an unbuilt tree partly; effect statements do not.

4.5 Multi-targeting

A project with several target frameworks (`<TargetFrameworks>net8.0;netstandard2.0</TargetFrameworks>`) is loaded by Roslyn **once per target framework**. The same source file is therefore analyzed as many times as there are frameworks. Whether that costs much is solution-specific: on some repositories the duplicates are the majority of all documents, on others they are negligible.

By default AICB analyzes all instances. You can opt into analyzing only the preferred instance:

```
{
  "analyzePreferredTfmOnly": true
}
```

in the solution's `.aicb.json` . The preferred instance is the one with the newest target framework; a tie keeps the first in load order. The setting lives only in the sidecar - there is no equivalent database setting.

What the switch does and does not change:

- **The type and method inventory is unchanged.** The query surfaces deduplicate to one logical project instance per name anyway, keeping the newest target framework's instance.
- **Lost:** a fan-in edge whose only source is a non-preferred instance. This affects `find_usages` , `impact_of_change` , `call_graph` , the deprecated-API insight, PageRank and the trimming priorities.
- **Gained:** more complete transitive side-effect facts, because the narrower run avoids a last-write-wins effect in the shared method index.
- `find_dead_code` stays safe either way: it requires an analyzed caller list and abstains where the edge went missing, instead of calling a live method dead.
- The reported project count counts project *instances*, so a multi-targeted project contributes one entry per target framework.

4.6 How long it takes

There is no universal formula of the form "solution size → analysis duration"; the time depends on the number of projects and documents, on restore state and on disk speed. The published benchmark in the architecture, limits and evidence guide reports measured cold analysis and warm-query times on three public solutions (≈ 25k, ≈ 55k and ≈ 1.8M lines of C#): about 10 s, 30 s and 97 s cold, warm answers in well under 2 s. Two practical statements hold:

- The analysis itself is usually a matter of minutes even for large solutions.
- Producing the **full Markdown export** of a very large solution is the expensive step, not the analysis: it builds the document single-threaded and can take considerably longer than the analysis it renders.

For everyday work on large solutions, prefer the bounded views - `architecture_overview` for orientation, `get_context`, `explain_symbol` or `prepare_task` for one symbol or one task - and reserve the whole-solution export for when you really need it.

After an edit, a refresh does not necessarily re-walk everything: where the previous run's facts can be reused for documents the edit could not have reached, they are - also when a change to a project or other non-C# file makes the refresh reload the solution through MSBuild. A `refresh_session(force: true)` reuses nothing. The refresh answer reports which path it took (`mode: incremental` or a full reload) and how many documents it reused. Adding, renaming or removing a C# file usually does not reload the solution either: the file is added to the loaded solution directly, and only where AICB cannot tell which project compiles it does the refresh fall back to a full reload. An edit to a `.cshtml` or `.razor` file counts as a change of the solution too and takes the full reload, as does adding or removing a `.xaml`, `.razor`, `.cshtml` or project file.

4.7 Side-effect classification

The analysis classifies every method by what it touches in the outside world. The vocabulary is closed and has exactly eight tokens:

Token	What it asserts
`io`	Filesystem and stream access
`network`	HTTP, socket, mail and cloud-storage calls
`database`	Database drivers and ORMs (for example ADO.NET/ `System.Data`, EF Core, Dapper, `Microsoft.Data.Sqlite`, Npgsql, MongoDB, Cosmos)
`serialization`	(De)serializer calls (for example `System.Text.Json`, Newtonsoft.Json, `XmlSerializer`, DataContract, MessagePack, protobuf, YamlDotNet)
`logging`	Logger calls (for example `Microsoft.Extensions.Logging`, Serilog, NLog, log4net)
`cache`	Calls into an **external** cache API (for example `Microsoft.Extensions.Caching`, StackExchange.Redis, EasyCaching, FusionCache)
`messaging`	Message-bus and queue clients (for example MassTransit, RabbitMQ, Azure Service Bus/Event Grid/Queues/Notification Hubs, Kafka, NATS, MediatR, Amazon SQS/SNS)
`unknown`	An external call no classification rule matched - reported so the gap is visible instead of reading as pure

The criterion is contact, not topic. This distinction is the one readers most often assume the other way:

- A `System.IO` type is judged on whether it touches a disk or a handle. Types whose members only transform values in memory carry no effect: `Path` (path strings), `FileSystemName` (glob matching), `MemoryStream`, `StringReader` / `StringWriter`. Types that reach a disk or a handle do: `File`, `Directory`, `FileStream`, `FileInfo`, `StreamReader` / `StreamWriter`.
- A `System.Net` type is judged the same way. Addresses (`IPAddress` / `IPEndPoint` / `DnsEndPoint`), cookie and header containers, credential holders, the HTTP and mail *message* surfaces and all of `System.Net.Mime` carry no effect; `Dns`, `Sockets`, `WebClient`, `SmtpClient`, `HttpClient`, `HttpContent` and third-party clients (Flurl.Http, Azure Blob Storage, Amazon S3/SES) do.
- A cache a type holds **in memory** is not an outside-world effect and is never classified as `cache`. Find such types through their cache interface (`find_implementations`) or their name (`find_symbol`).

Effects propagate. A method that calls a method with `io` carries `io`, and so does a method that builds an object whose constructor has `io`. Propagation follows calls and creations upward, including a call through an interface when exactly one implementation exists; a call with several possible implementations stays unresolved. A constructor that chains to another with `: this(...)` or `: base(...)` carries that constructor's effects, and a field initializer counts toward the constructor that runs it. Because of this, a single misclassified external call would fan out over the entire caller tree - which is why the classifier is conservative about the types listed above.

> **Note:** effect, leak and external-call results require a restored/built solution. In an unbuilt tree they can be empty or incomplete without looking wrong.

The tool `find_by_side_effects` exposes this layer. It accepts `effect=<token>` (validated against the eight tokens - anything else is rejected, not answered with zero), `purity=pure|impure|any`, `scope` and `includeTests` (default `false`). With no effect and no purity filter, only impure methods are listed. A known token with zero hits returns a note quoting what the token means, and a note also reports how many methods a filter removed, so a short list is never silently short.

4.8 Layer profiles and architecture analysis

AICB checks the Clean/Onion direction rule: dependencies should point inward. The check has its own vocabulary of five layer names, ordered from the inside out:

```
Contracts (0)  <  Domain (1)  <  Application (2)  <  Infrastructure (3)  <  Presentation (4)
```

`Contracts` is the innermost shared kernel (referenceable by everything, referencing nothing outward); `Presentation` is the outermost layer. **A violation exists when the source layer's rank is lower than the target layer's rank** - an inner layer depends on an outer one.

A type's layer is resolved in this order:

1. A real `<ai layer="...">` annotation.
2. The **namespace profile** configured for the solution - the `layerRules` in `.aicb.json` (or the profile stored in the configuration database), passed to the analysis.
3. The **project/assembly name**, by a generic Clean-Architecture convention. The assembly is the reliable layer signal; a type labeled this way counts as authoritative.
4. The **role heuristic** - the weakest fallback.

A profile may use its own layer names. Those names are ranked through the same convention, matched as whole dot-separated segments:

Name in the profile	Ranks as
`Core`, `Domain`, `Entities`	Domain
`Persistence`, `Adapters`, `DataAccess`	Infrastructure
`Application`, `UseCases`	Application
`Contracts`, `Abstractions`	Contracts
`Composition`, `Probes`, `Web`, `Api`, `Ui`, `Cli`, `Console`, `Mcp`, `Host`, `Samples`, `Benchmarks`, ...	Presentation

Name in the profile	Ranks as
anything else, for example `CodeAnalysis` or `DomainModel`	no rank

A layer name that is an outer-ring host word **dominates** an inner-layer word in the same name: a project named `MyApp.Core.Benchmarks` resolves to Presentation, not Domain. That is the safe direction - an outermost host's outbound edges are always legal.

The rank decides the direction only; the reported violation line prints the profile's own layer label, so you can see which label the name bought.

Coverage is disclosed. The layer check classifies every production type into one of three groups, and the insight reports them:

Group	Meaning
Checked	The type's layer has a rank; its dependency edges were judged
Unranked	The type has a layer whose name has no position in the Clean/Onion order - reported separately, with the layer names and type counts
Unresolved	No layer could be assigned at all

The skipped types are named (the first few inline), so a clean result over a mostly skipped solution cannot be read as "mostly clean". The insight also suggests how to fix it: rename the unranked layer to one of the five names or to a name built from their vocabulary, or add namespace-to-layer rules for the unassigned types.

Two **deliberate false negatives** keep the check quiet rather than noisy:

- **Generics:** dependency strings of generic types carry their type arguments (`IRepository<Order>`), while the index key is the bare identifier (`IRepository`) - such edges are not matched.
- **Intra-assembly:** an edge between two types of the *same* project is not a violation, provided neither side has an authoritative layer. Rank differences produced by the role heuristic inside one assembly are noise.

Production focus: types from test projects are skipped - a test referencing its Infrastructure target class is not a production layer violation.

> **Note:** a layer whose name sounds like an outer ring - `Api` , `Host` , `Adapters` , `Web` - is ranked as Presentation. That is correct when the name means the outer ring, and a false positive when `Api` is a library's public surface. The insight names the layer and the profile so you can see which rank the name bought.

Where you meet this layer:

- The `quality-layer-violations` **insight** in the insights list. Its severity follows the profile's layering policy: `Advisory` (the default) reports `Warning` , `Strict` reports `Critical` . The insight can be disabled in `Settings → Quality Profiles` .
- The **policy gate** (`evaluate_change_set`), which asks before an edit whether it would introduce a layer violation, with delta semantics against the current session.

To configure the layer axis, use the guided setup (`init_solution_config` then `apply_solution_config`), which writes the rules into the configuration database and into the git-tracked `.aicb.json` next to the `.sln` . A rule has a `pattern` , a `matchType` (`Contains` , `StartsWith` , `EndsWith` , `Exact`) and a `layer` . The sidecar's `layeringPolicy` key accepts `Advisory` or `Strict` . You can also edit the file by hand.

The same `.aicb.json` carries the **namespace exclusion list** (`exclusions` , same match types). Namespaces on that list are left out when the analysis builds type references - parameters, return types, fields, properties and dependency lists - so framework noise does not count toward coupling or dependency-based results.

4.9 Dead-code detection

`find_dead_code` answers "what is never used?" on two axes, selected with the `kinds` parameter: `method` (the default), `type` or `all` . An unrecognized token (for example the plural `types`) returns the superset of both axes rather than silently swallowing one.

The method axis reports a method only when both conditions hold:

1. It is **private**.
2. It has an **analyzed caller list**.

A private method that a framework may call is not judged either: `[RelayCommand]` or other framework attributes, partial methods, markup or designer event handlers, `static Main` , a record's `PrintMembers` , `ShouldSerialize` / `Reset` methods. A `nameof` in a method's attribute (`CanExecute`) counts as a call.

The exclusions are counted separately and reported: `methodsIneligibleNotPrivate` (the deliberate policy - public and internal methods have framework escapes such as reflection, DI and event wiring that static analysis cannot see), `methodsIneligibleFrameworkInvoked` (the private methods a framework may call, as above) and `methodsIneligibleUnanalyzedCallers` (private methods whose caller list was not analyzed - the blind spot, for example after unresolved references or for an explicit interface implementation). Merged into one number, you could not tell which arm emptied your scope. Committed `*.Designer.cs` and `*.g.cs` files are read for references but never judged. Test projects are excluded by default (`includeTests` opts in), and `testMethodsFiltered` reports what the filter removed.

The type axis reports types with no incoming reference in the reliable reverse type fan-in, minus the types that are structurally invisible to it. Four exemption families are applied and counted apart, so a zero over an eligible population is distinguishable from a scope that could not produce a candidate:

Family	Examples
Structural	Interfaces; static classes, including `*Extensions` containers
Test scaffolding	Test and fixture types, test paths, runner-activated types
Framework/reflection-activated	Types a framework instantiates by DI, reflection or convention - entry points, MVC controllers, middleware, EF migrations, Blazor pages, types with a non- `System` base or interface, routing/tool attributes, MEF parts (a type with an export attribute such as `[Export]` on itself or on a method, property or field)
Markup-referenced	Types referenced from XAML markup

The counts are `typesIneligibleStructural`, `typesIneligibleTestScaffolding`, `typesIneligibleFrameworkActivated`, `typesIneligibleMarkupReferenced`; `testTypesFiltered` reports what the test filter removed. When the method axis runs alone and finds nothing, the answer also reports `typesUnexamined` - how many types the type axis would have judged - so a bare "0 dead" is not read as an all-clear.

A type of a project that builds a library and that is visible outside it stays listed but carries `publicApi: true`, and a leading `publicApiNote` explains why: a consumer outside the solution - a package user, another repository - records no edge here. Settle whether the library ships before you delete such a type.

> **Note:** caller lists are statically resolved. Reflection, DI containers, serializers and external assemblies stay invisible. Absence of callers is not proof of dead code. Confirm a candidate before deleting it - the answer's leading note names the checks to run.

Both axes are capped at 200 entries; `scope` narrows the query to a namespace prefix.

4.10 Circular namespace dependencies

`detect_circular_dependencies` finds namespaces that depend on each other in a cycle. The granularity is the **namespace**, not the type: type cycles are common in C# and mostly harmless, project cycles are forbidden by the compiler, and the namespace level is the meaningful, low-noise module cut.

- A namespace edge `N → M` exists as soon as any type in `N` references a type in `M` (with `M ≠ N`).
- A reported cycle is a strongly connected component of at least two namespaces.
- Resolution is exact, by fully qualified name - same-named types in different namespaces are never conflated. References inside generic type arguments are decomposed (`C.Box<B.Bar>` counts as a dependency on both `C` and `B`). A reference that does not resolve to a solution type is external (BCL/NuGet) and is skipped.
- Each cycle names the **assemblies** its namespaces are declared in. A cycle *inside* one assembly is pure namespace organization; a cycle that *spans* assemblies means a namespace is reused across DLL boundaries - the more interesting situation, since project-reference cycles cannot compile. Cycles that span assemblies are listed first, then the larger tangles.
- Each cycle lists witness edges - one concrete type-to-type reference per directed namespace pair - and reports how many distinct type references cross that boundary. Read the witnesses as examples: where the count is higher than one, fixing the shown reference leaves the namespace edge standing.

`scope` narrows the report to cycles touching a namespace prefix; the graph itself stays the whole solution, so the scanned-namespace count is unchanged. Test projects are excluded by default (`includeTests` opts in); multi-targeting collapses by itself, because namespaces and edges are sets. Up to 100 cycles are returned, with at most 50 witness edges per cycle.

4.11 Test detection and coverage gaps

What counts as a test project. A project is classified as a test project when either signal holds:

- An explicit `.csproj` marker: an `<IsTestProject>true</IsTestProject>` property or a `Microsoft.NET.Test.Sdk` package reference. This is name-independent and always wins.
- A project-name rule from the active test profile. The default profile matches any project name containing `Test`, `Tests`, `Spec` or `Specs` (case-insensitive).

A multi-targeted instance is matched on its logical name too, so a suffix such as `(net8.0)` does not break a rule.

What counts as a test case. Only a method that itself carries a test attribute. A fixture builder or helper in a test project is not a test case. The attribute names come from the active test profile:

Profile	Test attributes
Default (framework-agnostic)	`Fact`, `Theory`, `Test`, `TestMethod`, `TestCase`
xUnit	`Fact`, `Theory`
NUnit	`Test`, `TestCase`, `TestCaseSource`, `TestFixture`
MSTest	`TestMethod`, `DataTestMethod`, `TestClass`

Attribute names are matched as substrings, case-insensitive, so a custom `[WindowsFact]` matches via `Fact`.

Match tiers. `find_tests_for` reports each covering test with the strength of the evidence, in nine tiers. They are listed here in sort order:

Value	Meaning
`invokes`	The test calls the target itself - the strongest evidence.
`invokes-via`	The test reaches the target through one method or constructor it calls (its own helper, a production entry point, or an object it builds).
`accesses`	The test reads or writes the target property or field; for a type, a static one.
`accesses-via`	The same, one hop away.
`constructs`	The test builds the target type (type and constructor queries only).
`constructs-via`	The same, one hop away.
`dispatch`	The test calls an interface or abstract member the target implements - a test double may have run instead.
`dispatch-via`	The same, one hop away.
`name`	The test's or its class's name mentions the target - a guess, not evidence of a call.

The first six tiers are "strong". The evidence reaches two hops at most: the test and one method or constructor it calls, or one property getter it reads. `invokesTotal` counts the strong rows and `dispatchTotal` the contract rows; the rest are name guesses. When a non-empty answer has no strong row at all, a `note` says so, so a full page of name guesses is not read as coverage. Stronger evidence sorts first, so the 50-entry cap can only ever drop the least load-bearing end.

Three query targets can all produce the same empty list, and the answer distinguishes them:

- **Not declared** - no declared type and no declared member carries the name. An empty answer then says nothing about coverage.
- **Type, method, operator, property or field** - what the strong tiers can index (properties and fields through the `accesses` tiers).
- **Event or enum member** - resolvable for `find_usages` and `impact_of_change`, but only the name tier can match it.

Coverage gaps. `coverage_gaps` lists production code without strong test coverage, per scope, capped at 200 entries. It is a **one-hop list, not a coverage measurement**: a method that a test reaches through another method is still listed. Each entry therefore carries `testReachDepth` - the number of hops to the nearest directly tested method. An entry *without* a depth is one no test reaches at all; those are the real gaps to work first. A constructor is a hop of that walk: `new X(...)` reaches the constructor it binds and what that constructor calls, so a method only a constructor calls gets a depth too.

The population is disclosed, so a gap count can be read against its denominator: `judgedUnits` (the non-test, non-abstract methods the list is drawn from) out of `symbolsScanned`, plus `testProjectsExcluded` and `testFixtureTypesExcluded`. Abstract declarations (interface methods without a default implementation, abstract class methods) are excluded - a test covers the implementation, not the declaration. A type that declares a test-attributed method is treated as a test fixture, so its attribute-less `SetUp`, lifecycle and helper members - its constructors and its field and event initializers included - count as test code, not as uncovered production code; this also works for a fixture in a project whose name matches no test rule.

> **Note:** the absence of a detected test is not proof that no test exists. The list is built from statically resolved calls.

4.12 Dependency injection resolution

`resolve_injection` resolves Microsoft.Extensions.DependencyInjection registrations and MEF export attributes. It is a syntactic scanner with a best-effort semantic model, so it yields names even when the DI assembly itself does not resolve; a missing semantic model never aborts the scan.

A call is recognized as a registration when its method name matches:

```
^(Try)?Add\w*(Singleton|Scoped|Transient)$
```

The wildcard in the middle admits wrapper forms such as `AddKeyedSingleton` and project-local helpers such as `AddCachedSingleton`, while names like `AddTransientHttpErrorPolicy` stay out. Without this, a helper's call sites would not count as registrations at all.

Beyond the method name, the scanner also reads:

- **Descriptor adds:** an `Add`, `TryAdd`, `TryAddEnumerable` or `Replace` that hands the collection a `ServiceDescriptor` built at the call or in a local there (`new ServiceDescriptor(...)`, `ServiceDescriptor.Singleton`, `Scoped`, `Transient` or `Describe`). A `TryAdd`, `TryAddEnumerable` or `Replace` row names the call in its `note`.
- `AddHostedService<T>()`, read as `IHostedService` implemented by `T`, with lifetime `Singleton` and the note `AddHostedService`.
- **Project-local registration helpers of any name:** a method of the solution that registers its own type arguments is read at each of its calls, with that call's own type arguments and key. Such a row carries the note `via <helper>`, and the helper's own body line is not listed again where a call read through it is in the answer.
- **Package helpers read by their call shape:** a chain such as `AddSingletonAs<T>().As<I>()` from a package, whose body the scanner cannot see, is read as "registers `I`, implemented by `T`", with the lifetime its name says. Such a row carries the note `inferred, not read` and never takes part in `ambiguous` or names a winner in `effectiveRegistration`.

- **MEF exports:** an `[Export]` of `System.ComponentModel.Composition` or `System.Composition` (or an attribute derived from their `ExportAttribute`) on a class or on one of its methods, properties or fields. The row's service is the `typeof` argument, otherwise the part or the exported property's or field's type; a method export without a `typeof`, and a derived attribute without one, give no row; a contract name becomes its `key`, and a member export's implementation reads `(factory)`. The lifetime follows `[PartCreationPolicy]` (MEF 1: `Shared` is `Singleton`, `NonShared` is `Transient`, otherwise `(dynamic)`) or `[Shared]` (MEF 2: `Singleton`, otherwise `Transient`). Such a row carries the note `MEF export`; it never makes `ambiguous` true and never names a winner in `effectiveRegistration`, because the MEF container composes by contract and not in a registration order.
- **Umbraco collection builders:** `Append`, `Insert`, `InsertBefore`, `InsertAfter`, `Add`, `Remove`, `Replace`, `Exclude` and `Clear` on a builder derived from Umbraco's `CollectionBuilderBase` are listed in writing order under `collectionBuilders` when the query names the collection, its item type, the builder or a collected type. They are not registration rows: what the collection holds at runtime depends on the order the composers run in.

For each registration the answer reports the service type, the concrete implementation, the lifetime (`Singleton`, `Scoped`, `Transient`), the location (`file:line`), the project and the declaring method. The service type keeps its type arguments (`IDisplayDriver<ISite>`, `IHandler<>` for an open registration), and a key or a closing makes a service of its own. A registration whose service type cannot be read statically is disclosed with one of three placeholders:

Placeholder	Meaning
`(dynamic)`	One side of the registration is computed at runtime - real, just not readable here
`(unknown)`	A call shape the scanner did not parse
`(factory)`	A factory lambda whose body does not name the type it produces

At most five unreadable sites are named; beyond that the list is abbreviated. One data-driven shape **is** expanded: a `foreach` over a static table of `new(typeof(Service), typeof(Impl))` rows becomes one registration per row, even when the table lives in another project. Rows produced at runtime - reflection or assembly scans, method results, a deconstructed loop variable - are not expanded and are disclosed instead.

Further resolution details:

- `site` narrows to registrations whose file path contains a given substring.
- `includeTests` (default `false`) excludes registrations declared in test projects; `testRegistrationsFiltered` reports what was hidden.
- A closed generic query (`IHandler<Saved>`) lists the registrations of that closing and the open registrations (`IHandler<>`) that serve every closing; `typeArgumentFilter` counts the registrations closing it otherwise (`otherClosings`) and those whose closing the call does not fix (`unresolved`). `IHandler` or `IHandler<T>` lists every closing.
- `ambiguous` is judged within one registration site, per key and per closed service type: true only when that site registers more than one distinct, statically resolvable implementation and the service is not consumed as a collection. Rows that add to the service's collection by design (`TryAddEnumerable`, `AddHostedService`) are no conflict. Across sites it is deliberately not called ambiguous - a solution with several hosts registers the same service once per host, and those containers never meet. `multiRegistration` and `registrationSites` tell you whether more than one registration matched at all.

- `consumedAsCollection` is true when the service is consumed as a collection - a `GetServices<T>()` call or an `IEnumerable<T>`-family (or `T[]`) constructor parameter, optional and nullable ones included. Multiple registrations are then a deliberate multi-binding, not a last-wins conflict. A consumer declared in a test project counts only under `includeTests`.
- `otherConsumers` counts the consumers that take the service other than through a constructor parameter, one entry per form, and is omitted when there are none: `locator` - a `GetService<T>()`, `GetRequiredService<T>()` or `GetService(typeof(T))` call or its keyed twin, on any receiver - and `inject-property` - an `[Inject]` property in C#, a Razor `@inject` directive or an `[Inject]` property in a `@code` block. Each entry carries `consumers`, `sites` and up to three `examples`; test projects count only under `includeTests`. `constructorConsumerCount` keeps counting constructor parameters only, and `GetServices<T>()` stays with `consumedAsCollection`.
- An empty answer is distinguished in three ways: a name that resolves to nothing is reported as not found with a nearest-name suggestion; a real name with no registration gets a note stating what the scanner reads and naming the forms it does not read, together with which of them the solution actually holds: helpers whose body is not in the solution (a package's, such as `AddHttpClient`, or one a source generator writes), registration by convention or assembly scan (Scrutor), a `ServiceDescriptor` that is not built at the call or in a local there, types decided at runtime, MEF parts exported through `[InheritedExport]` on an interface or through a convention builder (an `[Export]` attribute is read), and Autofac, Castle Windsor and other non-Microsoft containers. An empty list therefore means "not statically visible", never "not registered". If the solution references a foreign container assembly, the note names the projects and assemblies instead of the generic list.

Instantiation sites. `instantiation_sites` answers the narrower question "who creates an instance of this type?" with two edge kinds kept apart:

- **created** - a method, constructor or accessor that runs `new X(...)`, or a file of top-level statements that does, named by its label (`<path>/Program.cs (top-level)`) with no declaring type or namespace, so a namespace `scope` leaves it out. This includes target-typed `new`, `record with` expressions and collection/`stackalloc` allocations. A field or property initializer is attributed to the constructor that runs it (the implicit one when none is declared); with several non-chaining constructors, one initializer line yields one site per constructor.
- **injected** - a type that receives `X` as a constructor-injected dependency.

The answer reports `createdByCount` and `injectedIntoCount`, and it includes test-project sites, because a test that constructs a type breaks when the constructor changes. A concrete type registered in a DI container correctly reports zero injections: its consumers take the interface, so the edges sit on that interface name - the answer names the interfaces in that case. A type name that matches no declaration is reported differently, with a nearest-name suggestion rather than the zero-explanation note.

4.13 PageRank and the relevance score

AICB contains a personalized PageRank calculation over the method graph. Its edges come from the caller lists, inverted (caller → callee), and its parameters are fixed:

Parameter	Value
Iterations	30, with no convergence check and no early exit, so the result is deterministic; a non-converged result is returned as is
Damping	0.85

Parameter	Value
Hint boost	Methods recognized in the user prompt receive 10× more initial mass

Duplicate method keys (overloads, multi-targeted instances) collapse to one node, so the total mass stays approximately 1.

> **Note:** PageRank is implemented but not active. No released feature turns it on: the scoring pipeline passes a PageRank value of 0, so the factor below is 1.0 in every real run, and there is no setting to enable it. PageRank is also not a ranking factor of the search tools.

What actually ranks methods today is the relevance score used when a token budget is applied to a rendered context. The score is multiplicative; the lower it is, the sooner the method is trimmed away. Its factors are:

Factor	Value
Log-damped fan-in	`1 + log(1 + caller count)`
PageRank boost	`1 + 5 × pageRank` (currently always 1.0 - see above)
Visibility	`public` 2.0 · `internal` 1.0 · `protected` 0.7 · `protected internal` 0.85 · `private protected` 0.5 · `private` 0.3
Priority (`priority` annotation)	`high` / `important` 3.0 · `low` / `nice-to-have` 0.2 · otherwise 1.0
Entry point	5.0
Role	Controller 2.0 · Helper/Utility 0.6 · otherwise 1.0
Path penalty	Test 0.1 · Generated 0.05 · Designer 0.05 · Migration 0.3
User hint	Methods recognized in the user prompt get a multiplier of 10
Dead private	`private` and no callers → 0.1
Single-caller private	`private` and exactly one caller → 0.4

All multipliers except the two compound visibility values (`protected internal` , `private protected`) can be overridden by the active compression-rules profile; without a profile the values above apply. Scores are rounded to nine decimals, and a multi-stage tiebreaker (score, lines of code, method name, method key) keeps the ordering deterministic.

The token-budget machinery is active only when it is switched on - the built-in `Default` pipeline profile has token trimming off, and the `Aggressive Trimming` profile turns it on with a 20,000-token budget. Context exports that are not budgeted are not affected by the score.

4.14 Metrics

Metrics are computed per method, per type, per namespace and per solution.

Per method

Metric	Meaning
Cyclomatic complexity	McCabe complexity - the number of independent paths. `0` means not computed
Lines of code	The lines of the declaration that carry a token (signature plus body; blank and comment-only lines excluded). Empty means not computed
Parameter count	Number of parameters

Per type

Metric	Meaning
Efferent coupling (Ce)	Number of distinct referenced types. Framework types are included unless removed by the namespace exclusion list
Afferent coupling (Ca)	Number of types that use this type
Member count	Methods + properties + fields, without constructors
Method count	Number of methods

> **Note:** the stored afferent-coupling field is empty on live analyses. The live surfaces that show Ca - the quality-gate metric `ca-max`, the `QUALITY_HOTSPOTS` section and the `ca=""` tag attribute - compute the reverse fan-in from the dependency lists instead. The field remains valid for hand-built models and older snapshots.

Per namespace - Martin's package metrics, shown in the `QUALITY_HOTSPOTS` section of a context document:

Metric	Meaning
Namespace / layer	The namespace and the architectural layer its types resolve to; `(mixed)` when its types disagree, `(global)` for types without a namespace
Type count	Logical types in the namespace (partial fragments and multi-targeted instances unified)
Abstract type count	Interfaces and abstract classes - extension points a consumer can depend on without depending on an implementation
Efferent coupling (Ce)	Distinct type names *outside* the namespace that types inside depend upon
Afferent coupling (Ca)	Distinct types *outside* that depend on something inside
Instability (i)	`Ce / (Ca + Ce)`, from 0 (only depended upon) to 1 (only depends on others); 0 when the namespace has no coupling
Abstractness (a)	`abstract types / all types`, from 0 to 1
Distance from the main sequence (d)	`|a + i − 1|` - 0 means the namespace is as abstract as it is stable. High `d` means either unstable *and* concrete (hard to change, nothing to extend) or stable *and* abstract (an abstraction nobody uses)

Metric	Meaning
Has coupling	Whether the namespace has any coupling at all. Namespaces without coupling are filtered out of the ranking, because they would otherwise crowd the top of the list

The ranking orders by distance descending; ties are broken by total coupling and then by name. Ca is a lower bound for a library whose real callers live outside the analyzed solution, and Ce counts the types the analysis recorded in signatures and explicit generic arguments - a low instability means "no *recorded* outward coupling", not proof of a stable namespace. In a partial (budget-trimmed) export the main-sequence block is omitted entirely, because pruning would bias the numbers toward a false all-clear.

Per solution - the roll-up behind `solution_metrics` and the quality gate, computed over production code only (test projects excluded; the tool reports the excluded test-side counts as well):

Metric	Meaning
Type count	Logical types - partial fragments merged, multi-targeted instances deduplicated
Method count	Declared methods (accessors, constructors and operators are not counted here)
Complexity max	Highest cyclomatic complexity over all methods (`0` = none computed)
Complexity avg	Average complexity over the methods whose complexity was computed
Methods over complexity threshold	Number of methods with complexity **strictly greater** than the configured threshold
Ce max	Highest efferent coupling over all logical types
Ca max	Highest afferent coupling, computed as reverse fan-in from the dependencies

Two complexity axes. `symbol_metrics` reports both cyclomatic complexity and SonarSource-style **cognitive complexity** - how hard a method is to understand, with a nesting penalty, `else` / `catch` counted and boolean-operator runs collapsed. You choose the ranking axis with `rankByCognitive` (default: cyclomatic). The two axes can disagree about *which* methods appear, not merely about their order: a flat method with many independent paths can top the cyclomatic list while a genuinely tangled method is absent from it entirely, and the other way round. When the axes disagree about the set of methods, the answer carries a note naming the axis that produced the ranking and the strongest methods the other axis would have shown. A `minComplexity` floor is **inclusive**: a method sitting exactly on the floor is listed.

> **Note:** the strict and inclusive boundaries are both correct and intentionally different. `MethodsOverComplexityThreshold` in the solution roll-up counts strictly greater, while `symbol_metrics(minComplexity)` and the complex-untested insight list a method sitting exactly at the threshold. At the same number the inclusive lists are larger by exactly the boundary methods.

5 The context document (AI-Builder-MD)

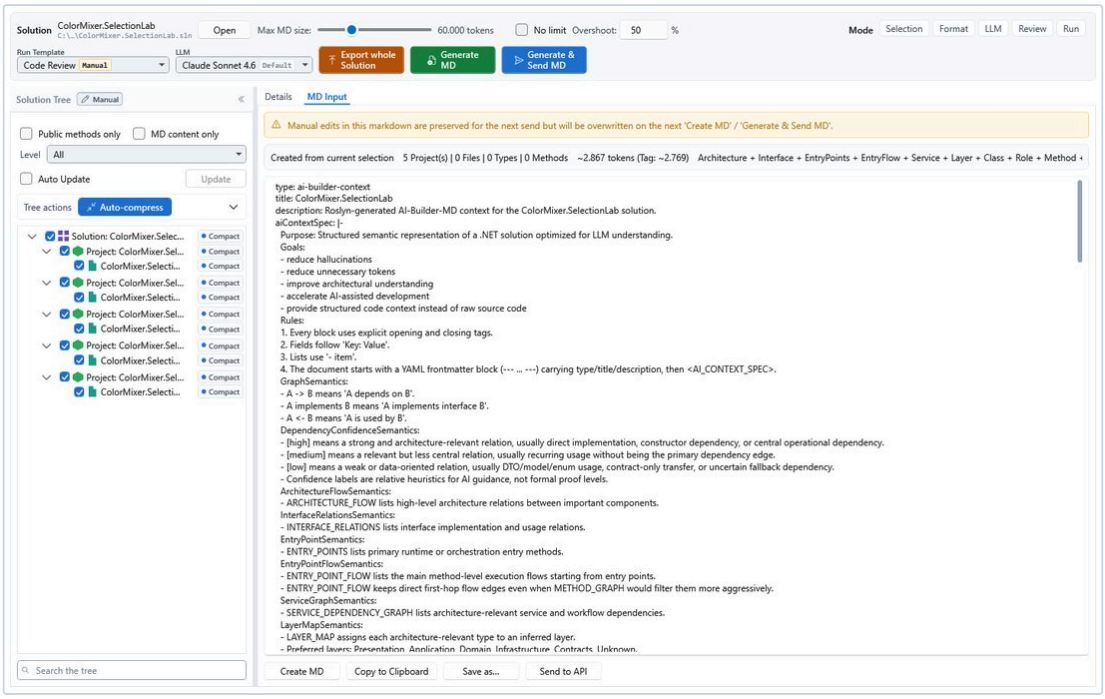
The context document is the deliverable of AIContextBuilder: a single text file that describes one analyzed .NET solution in a structured, compact form. It is written for language models, but it is deliberately readable for humans as well. Instead of raw source code, it contains the solution's structure (projects, files, types, members), its architecture (dependency and call graphs), the resolved semantics and developer annotations, plus the supporting build and configuration files.

This chapter explains the file itself: its name, the two notations it can be written in, every section it contains, the legends that decode it, the detail and compression levels, and the token budget that limits its size.

5.1 The file

The document is written as one UTF-8 file per solution. `{SolutionName}.analysis.ai.md` is the automatic exporter convention; the GUI's `Save as...` action proposes `{SolutionName}-context.md`, while CLI and MCP exports use the path supplied by the caller. The extension stays `.md` regardless of the notation used for the content.

The same solution can be rendered more than once with different settings - for example through a different context template, a different MD profile or a different token budget. Each render produces a complete document of its own.



A rendered context document in the desktop app's MD Input tab

5.2 The two notations

The content of the document can be written in two notations. The information, the section order and the field values are identical in both; only the way they are written changes.

Notation	Value	Description
Tag	`tag`	The established AI-Builder format: every block is wrapped in explicit `<TAG>` and `</TAG>` lines, fields are written as `Key: Value`, lists as `- item`.

Notation	Value	Description
YAML	`yaml`	The same content as idiomatic YAML. This is the product default.

Select the notation with the `--format tag|yaml` option of the CLI, with the `format` parameter of the MCP tool `export_markdown`, or through the output format configured on a run template or on the active MCP profile. The GUI exposes the same switch.

Two details are worth knowing:

- `Tag` is the numeric zero value of the notation setting and therefore the technical fallback wherever no notation is configured. It is **not** the product default - that is `yaml`.
- The CLI rejects an unknown `--format` value with an error message (`--format must be 'tag' or 'yaml'`). In `export_markdown`, an unrecognized `format` value falls back to the tag notation; because any supplied value counts as explicit, it also overrides the notation configured on the active MCP profile.

Note: YAML is a re-notation, not a different selection of content, and it is not promised to be cheaper. Whether it needs fewer tokens than the tag notation depends on the individual document; both notations carry exactly the same information.

How the YAML notation is shaped

The YAML rendering follows a fixed set of rules:

- The frontmatter fields (`type`, `title`, `description`) become top-level keys.
- Every block becomes a mapping key in lower camel case: `<SOLUTION>` becomes `solution:`, `<AI_CONTEXT_SPEC>` becomes `aiContextSpec:`, `<BUILD_AND_UI_FILES>` becomes `buildAndUiFiles:`.
- Repeated blocks become sequences under a plural key: `PROJECT` becomes `projects:`, `FILE` becomes `files:`, `CLASS` becomes `classes:`, `INTERFACE` becomes `interfaces:`, `ENUM` becomes `enums:`, `METHOD` becomes `methods:`.
- A list item that carries a confidence marker in the tag notation (`- IWidgetStore [high]`) becomes a nested mapping with `name` and `confidence` keys.
- Multi-line content (source code, the format contract, a verbatim file body) becomes a block scalar introduced by `|`. When the first content line of such a block already starts with a space, the explicit indicator `|2` is used so the indentation stays unambiguous.
- Block attributes such as `compact="isolated"`, `lines="10-20"` or `path="App/Widget.cs"` are carried over as regular fields.
- The format contract of the tag notation is not split into YAML keys. It is a description of a different notation, so it is preserved as one string field.

Values that could be misread as numbers or booleans are quoted so they stay strings - for example `version: "1.0"`. The transformation is lossless: no section, no field and no code line is dropped.

5.3 The document header

Every non-lean document starts with three blocks that describe the document itself.

YAML frontmatter

```
---
type: ai-builder-context
title: Acme.Solution
description: Roslyn-generated AI-Builder-MD context for the Acme.Solution solution.
---
```

The frontmatter makes the document indexable for tools that read YAML frontmatter. `title` is the solution name (or `solution` when the name is empty). The frontmatter is written when the document is not lean **and** the format contract section is actually part of the document - if that section is switched off by a profile, the frontmatter is omitted with it.

AI_CONTEXT_SPEC

`AI_CONTEXT_SPEC` is the format contract. It tells the reader how the rest of the document is written and is present in full exports:

- `Purpose` and `Goals`: what the document is for.
- `Rules` 1-4: every block uses explicit opening and closing tags; fields follow `Key: Value`; lists use `- item`; the document starts with the YAML frontmatter followed by `AI_CONTEXT_SPEC`.
- `AliasRules` 5-7: `PATH_LEGEND` maps type names to short aliases; aliases may replace type names **only** in ten named sections; `PATH_LEGEND` always contains the full mapping. See "PATH_LEGEND and alias rules" below.
- `GraphSemantics`: `A -> B` means "A depends on B"; `A implements B` means "A implements interface B"; `A <- B` means "A is used by B".
- `DependencyConfidenceSemantics`: the meaning of the confidence markers `[high]`, `[medium]` and `[low]`.
- One semantics paragraph per included graph section (for example `LayerMapSemantics` or `MethodUsedByGraphSemantics`).
- `SemanticModel`: which sub-blocks a type block and a method block may contain, where the values come from, and that empty blocks and empty fields are omitted.
- `Sections`: the list of sections contained in **this** document.

Note: the `Sections:` list is document-specific, not a static contract. It lists only the sections that this particular render produced, so a document with a switched-off section or without XAML files names fewer sections than a full export. Use it as a table of contents for the document in front of you, not as the definition of the format.

AI_CONTEXT_META

`AI_CONTEXT_META` identifies the producer and counts the content:

```
<AI_CONTEXT_META>
Generator: AIContextBuilder
Version: 1.0
Format: AIContextMarkdown
Solution: Acme.Solution
GeneratedAt: 2026-09-21T09:15:42Z
AnalyzerVersion: 0.5.464.xxx
Projects: 2
Files: 2
Types: 4
Classes: 2
Interfaces: 1
Enums: 1
Methods: 5
Properties: 2
Constructors: 1
DistinctDependencies: 3
</AI_CONTEXT_META>
```

Field	Meaning
`Generator`	Always `AIContextBuilder`.
`Version`	The document format version.
`Format`	Always `AIContextMarkdown`.
`Solution`	The name of the analyzed solution.
`GeneratedAt`	The render time in UTC.
`AnalyzerVersion`	The version of the analysis engine that produced the model.
`Projects`	Number of projects in the document.
`Files`	Number of source files.
`Types`	Total number of types.
`Classes`	Types that are neither interfaces nor enums.
`Interfaces`	Number of interface types.
`Enums`	Number of enum types.
`Methods`	Number of methods.
`Properties`	Number of properties.
`Constructors`	Number of constructors.
`DistinctDependencies`	Number of distinct type dependencies across all types.

When the document covers only part of the solution - a slice, or a render reduced by the token budget - the counts are written as `X of Y`, for example `Types: 4 of 37`. A plain number means the document covers everything.

AI_PROMPT

If the export was given a prompt, the prompt is reproduced as bullet lists under the headings `SystemRole`, `Instructions`, `Goal`, `Constraints` and `AdditionalContext`. Each field is split at line breaks; leading `-` or `*` characters are removed so the result is a clean list. Fields without content are omitted, and the whole section is omitted when no prompt was set.

5.4 The sections at a glance

The following table lists the sections in the order they appear in a full export. The right-hand column names the conditions under which a section can be missing.

#	Section	Content	Can be missing when
1	`SPEC`	Format contract (see above).	The section slot is switched off.
2	`META`	Producer, version, solution, counts.	The section slot is switched off.
3	`AI_PROMPT`	The export prompt.	The slot is off or all five prompt fields are empty.
4	`QUALITY_HOTSPOTS`	Most complex methods, most coupled types, namespaces furthest from the main sequence. Opt-in.	Quality metrics are not enabled (the default).
5	`PATH_LEGEND`	Type name to alias mapping.	The slot is off, layout is disabled, or there are no aliases.
6	`COMPRESSION_LEGEND`	Explanation of the compression modes, the provenance vocabulary and the caller lists.	Never - always rendered.
7	`DOMAIN`	Solution name and its core components with their roles.	The section slot is switched off.
8	`ARCHITECTURE_FLOW`	High-level relations between important components.	The slot is off or the graph is not enabled.
9	`INTERFACE_RELATIONS`	Interface implementation and usage relations.	The slot is off or the graph is not enabled.
10	`ENTRY_POINTS`	Primary runtime and orchestration entry methods.	The slot is off or the graph is not enabled.
11	`ENTRY_POINT_FLOW`	Method-level execution flows starting at the entry points.	The slot is off or the graph is not enabled.
12	`SERVICE_DEPENDENCY_GRAPH`	Architecture-relevant service and workflow dependencies.	The slot is off or the graph is not enabled.
13	`LAYER_MAP`	Type to inferred layer assignment.	The slot is off or the graph is not enabled.

#	Section	Content	Can be missing when
14	`CLASS_DEPENDENCY_GRAPH`	Architecture-relevant class and interface dependencies.	The slot is off or the graph is not enabled.
15	`ROLE_GRAPH`	Role-to-role dependencies such as `Controller -> Service`.	The slot is off or the graph is not enabled.
16	`METHOD_GRAPH`	Method-level call dependencies.	The slot is off or the graph is not enabled.
17	`METHOD_USED_BY_GRAPH`	Reverse method usage relations.	The slot is off or the graph is not enabled.
18	`ENUM_SUMMARY`	List of all enum types.	The slot is off or the solution has no enum.
19	`FILE_INDEX`	Directory map: per file its project, relative path, type and method count.	The slot is off or no project contains a file.
20	`BUILD_AND_UI_FILES`	`.csproj` of every included project and the relevant `.xaml` views.	The disk scan finds nothing.
21	`AUXILIARY_FILES`	Config, build, documentation and resource files.	No file passes the inclusion rule.
22	`SOLUTION`	Start of the nested structure tree.	Never - the head block is written even for a solution without projects.
23	`PROJECT`	One project of the tree.	Part of the tree; omitted when there is no project.
24	`FILE`	One source file of the tree.	Part of the tree; omitted when a project has no file.
25	`CLASS`	One class (including records and structs).	Part of the tree.
26	`INTERFACE`	One interface.	Part of the tree.
27	`ENUM`	One enum.	Part of the tree.
-	`QUALITY_FINDINGS`	Code-quality findings, symbol by symbol.	The document is lean, or the finding list is empty. It has no profile slot and trails the document.

A section can also be absent because the document mode removed it (see "Document modes"). The quality findings section is deliberately not part of the ordered section list; it is appended after the tree in a full export, and sorted alphabetically in the ordinal ordering described below. In a document that renders only part of the solution (a slice or a selection export), the findings are judged over the whole solution and listed for the types the document renders; a finding that names no type at all stays, while a single finding line that names no type - a dependency cycle, for example - is left out. The section then carries the comment `<!-- Judged over the whole solution; listed for the types this document renders. -->`.

Section order

The sections follow a reading order that is oriented on the use case: format descriptions and legends first, then the architecture graphs, then the file index and supporting files, then the type tree. An alternative ordering sorts the sections alphabetically by their section id. It exists to maximize the shared prompt prefix when a provider caches prompt tokens across calls. Note: the alphabetical ordering is implemented but not yet released as a user option - the standard export, GUI and MCP paths all use the reading order.

5.5 The architecture graphs

The ten graph sections describe the solution's architecture without repeating source code. Their edges use a small, consistent vocabulary:

- `A -> B` - A depends on B (architecture flow, service graph, class graph, method graph).
- `A <- B` - A is used by B (method used-by graph).
- `A implements B` - A implements interface B (interface relations).
- `A => Layer` - the inferred layer of type A (layer map).
- `Role -> Role` - a role-to-role dependency such as `Service -> Interface` (role graph).

Concrete examples of the rendered lines:

```
<ARCHITECTURE_FLOW>
Flows:
- CMS -> CBC
- MCUC -> ICMS
</ARCHITECTURE_FLOW>

<INTERFACE_RELATIONS>
- OR implements IO
</INTERFACE_RELATIONS>

<LAYER_MAP>
- IO => Domain
- OS => Application
</LAYER_MAP>

<ROLE_GRAPH>
- Service -> Interface
</ROLE_GRAPH>

<ENTRY_POINTS>
- DR.RunAll()
- LPUC.ExecuteAsync(string, CancellationToken)
</ENTRY_POINTS>

<METHOD_GRAPH>
- CBC.BlendAdditive(RgbColor, RgbColor) -> CNH.Normalize(RgbColor)
</METHOD_GRAPH>

<METHOD_USED_BY_GRAPH>
- CNH.Normalize(RgbColor) <- CBC.BlendAverage(RgbColor, RgbColor)
</METHOD_USED_BY_GRAPH>
```

`ENTRY_POINTS` lists the entry methods with their parameter list. `ENTRY_POINT_FLOW` adds the direct first hop for each of them, in the form `Caller(params) -> Callee(param types)`. `METHOD_GRAPH` and `METHOD_USED_BY_GRAPH` list complete call edges; the method graph mode configured on the active detail preset decides how much of the signature each side carries. A constructor call is an edge like any other: `new OrderService(repo)` appears as `OrderService.OrderService(IRepository)`. A constructor of an external type is not listed, and a non-public constructor is filtered like a non-public method. Outside its `Full` mode, calls within one type are omitted - the graph shows calls *between* types, so a chain that delegates internally first appears without that hop. `ENTRY_POINT_FLOW` deliberately keeps those first-hop edges.

An empty graph is rendered as a single `- none` line, so you can always tell "the section was rendered and is empty" from "the section is not in the document".

A full export enables all ten graphs. The bounded architecture overview enables the eight macro graphs and leaves out `METHOD_GRAPH` and `METHOD_USED_BY_GRAPH`; for method-level structure, query a specific symbol instead.

5.6 DOMAIN, ENUM_SUMMARY and FILE_INDEX

`DOMAIN` names the solution and lists its core components as `- {name} [{role}]`, one line per type that has a resolved role:

```
<DOMAIN>
Name: Acme.Solution
CoreComponents:
- IO [contract]
- OR [entity]
- OS [service]
</DOMAIN>
```

`ENUM_SUMMARY` lists the names of all enum types under an `Enums:` header. It is only written when the solution actually contains an enum.

`FILE_INDEX` is the directory map of the document. Each file contributes a block with four lines:

```
<FILE_INDEX>
[Project: Acme.Domain]
**RelativePath:** src/Domain/Order.cs
Types: 3
Methods: 4
</FILE_INDEX>
```

`Types` counts the types declared in the file, `Methods` the declared methods of those types.

5.7 BUILD_AND_UI_FILES

`BUILD_AND_UI_FILES` brings the build and UI files of the analyzed code into the document. It is always on and relevance-filtered:

- The `.csproj` of every included project is always written.
- A `.xaml` file is written only when it is tied to the included code slice - either its code-behind type is part of the document, or an included type references or binds it.

Each file becomes a `FILE` block whose attributes name the relative path, the file type and both sizes, followed by the file content:

```
<FILE path="Sample.UI/MainWindow.xaml" type="xaml" original_bytes="180" compressed_bytes="90">
... content ...
</FILE>
```

The content is compressed according to the detail level resolved for that file. A `.xaml` file defaults to the `Detailed` level, a `.csproj` to `Normal`; both keep everything that carries meaning while removing formatting noise. The `Source` level writes the raw file. Per-file levels can be set in the solution tree; without an explicit setting the file type default applies.

5.8 AUXILIARY_FILES

`AUXILIARY_FILES` collects the non-code supporting files of the analyzed projects. The following extensions are considered: `.json`, `.xml`, `.config`, `.resx`, `.props`, `.targets`, `.md`, `.txt`, `.editorconfig`, `.yml`, `.yaml`. Files inside `bin`, `obj` and `.vs` folders are skipped.

The inclusion rule is a hybrid:

- **Project-wide structure and build files are always included**: `.props` and `.targets` files, `.editorconfig`, `global.json` and `README.md`.
- **Other loose configuration and documentation files are included only when the export prompt names them** - by file name (for example `config.json`) or, when it is specific enough, by its stem (`config`). The match is case-insensitive and must hit a whole word, so `config` does not match `reconfigure`. All five prompt fields are searched.
- **Potentially sensitive configuration files are gated behind their own opt-in** (`IncludeSensitiveConfigFiles`, default off). This subset contains every `*.config` file (for example `web.config`, `app.config`, `nuget.config`) as well as files whose name is or starts with `appsettings`, `launchSettings`, `secrets`, `compose` or `docker-compose` in one of the formats `.json`, `.yaml` or `.yml`. These files routinely carry connection strings, API keys or credentials, so they are never included unless you enable the option explicitly.

Each included file is written as a `FILE` block with the same attributes as above. The default detail level is `Source`, which writes the raw content; per-file levels from the solution tree are honored.

5.9 The solution tree

`SOLUTION`, `PROJECT`, `FILE` and the type blocks form one nested tree:

```
<SOLUTION>
Name: Acme.Solution
Projects: 2
<PROJECT>
Name: Acme.Domain
<FILE>
Name: Order.cs
Path: src/Domain/Order.cs
<CLASS>
...
</CLASS>
</FILE>
</PROJECT>
</SOLUTION>
```

The `SOLUTION` head block always names the solution and the number of projects; the tree is then nested one level deeper per project, file and type. `PROJECT` carries the project name and, when known, the project file path. `FILE` carries the file name, the path relative to the solution root and - when set - the namespace.

If a file's code is part of the selection, a `FILE_CODE` block with the complete file content is written inside the `FILE` block, fenced as C#.

5.10 The type blocks

A type becomes one `CLASS`, `INTERFACE` or `ENUM` block. The type kind decides which sub-blocks are possible; every sub-block that has no content is omitted.

Sub-block	CLASS	INTERFACE	ENUM	Content
header	yes	yes	yes	Type name, layer and role.
`TYPE_INFO`	yes	yes	yes	Objective metadata extracted from the source.
`SEMANTICS`	yes	yes	yes	Resolved semantic values with their provenance.
`SUMMARY`	yes	yes	yes	XML documentation summary.
`AI_TAGS`	yes	yes	yes	Developer-provided annotations.
`STRUCTURE`	yes	yes	yes	Members and declaration facts.
`DEPENDENCIES`	yes	yes	no	Type dependencies grouped by relation.
`METHODS`	yes	yes	no	The method blocks.
`USED_BY`	yes	yes	yes	Types in this document that use this type.
`TYPE_CODE`	yes	yes	yes	Verbatim source of the type.

The header renders the type name in bold, followed by the layer and role as bracket fields when they are resolved:

```
<CLASS>
**TypeName:** Order
[Layer: Domain]
[Role: Entity]
```

TYPE_INFO

`TYPE_INFO` contains facts taken directly from the source, without interpretation. Fields without a value are omitted; flag fields are written only when they are true.

Field	Meaning
`Name`	Simple type name.
`FullName`	Namespace-qualified name.
`AssemblyName`	Name of the assembly the type is compiled into.
`Kind`	Type kind (class, interface, enum, ...).
`Accessibility`	Declared accessibility.
`Modifiers`	List of declaration modifiers.
`IsStatic`, `IsAbstract`, `IsSealed`, `IsGeneric`	Flags, written only when true.
`TypeParameters`	List of generic type parameters.

Field	Meaning
`Attributes`	List of applied attributes.
`BaseTypeFullName`	Base type.
`InterfaceFullNames`	Implemented interfaces.
`SourceFilePath`	File the type is declared in.

SEMANTICS and provenance markers

`SEMANTICS` contains the resolved semantic values of a type or method: `Role`, `Layer`, `Domain`, `Context` and - for types - `Responsibility`. Layer values honor the active layer-mapping profile, so they agree with `LAYER_MAP` in the same document. A method's `Context` takes into account the side effects it reaches through the project methods it calls, so a method that writes a file only through a helper reads like one that writes it itself; its `Role` describes the method's own work. An interface method takes its `Context` from its implementations. A value an author set with an `<ai>` tag still wins.

The last line of the block states where the values came from:

```
<SEMANTICS>
Role: entity
Layer: Domain
Source: Resolved
</SEMANTICS>
```

`Source: Resolved` means the values come from the resolved analysis. When no per-field provenance was recorded, the marker stands alone. When the fields of a block have different origins, the line breaks them down:

```
Source: Resolved (role=inferred, layer=ai)
```

The four tokens are:

Token	Meaning
`fact`	Deterministic, taken from the source.
`ai`	Explicitly asserted by a developer annotation.
`inferred`	Heuristic guess - low confidence, may be wrong.
`verified-absent`	The author confirmed that the field is empty.

Fields confirmed as empty are listed after the breakdown, for example `Source: Resolved (; verified-absent: sideEffects)`. Treat `inferred` values as hints, not as ground truth, and do not rely on them where correctness matters.

A compact `SEMANTICS` block (see "Method compression modes") writes the values as one pipe-separated line and carries an optional `src:` line with the same vocabulary. A compact block without a `src:` line is fully inferred.

SUMMARY and AI_TAGS

`SUMMARY` reproduces the XML documentation summary of the type or method. A single-line summary is written as one bold line; a multi-line summary becomes a heading followed by a fenced block, so the original formatting survives.

`AI_TAGS` contains the annotations a developer attached to the element - the explicit semantic metadata that takes precedence over heuristic inference. Types can carry `Role`, `Layer`, `Context`, `Domain`, `Priority`, `Complexity`, `SideEffects`, `Stability`, `Responsibility`, `Pattern`, `DependencyType`, `Determinism`, `DataAccess`, `Interaction`, `Validation` and `ErrorHandling`; methods carry `Role`, `Layer`, `Context`, `Domain`, `Priority`, `Complexity`, `SideEffects` and `Stability`. Only the fields that are set are written, and the block is omitted entirely when nothing is annotated.

STRUCTURE

`STRUCTURE` describes the declaration itself. The header fields are `Access`, `BaseType`, `TypeKey` (the fully qualified identity) and `Ifaces`. Depending on the type kind, member lists follow:

```
<STRUCTURE>
Access: public
BaseType: Acme.Domain.Entity
TypeKey: Acme.Domain.Order
Ifaces:
- IOrder
Ctors:
- Order()
Props:
- Guid Id
- OrderStatus Status
</STRUCTURE>
```

Constructors appear under `Ctors:`, properties under `Props:`, fields under `Fields:` and enum members under `Members:`. Each member line carries its accessibility and, where applicable, modifiers such as `static`, `readonly` or `const`. When the detail settings ask for inferred values, a property or field with an initializer also shows `= <initializer>`; with the additional value-source annotation the line names where the value comes from, for example `(from property-init)`, `(from field-init)` or `(from const-definition)`. Method parameters with a default value show `= <value>` and, when enabled, the annotation `(from method-param-default)`.

DEPENDENCIES

`DEPENDENCIES` groups the type's relations into five lists:

Group	Meaning
`Implements`	Interfaces the type implements.
`DependsOn`	Dependencies the type receives or holds.
`Contracts`	Data- and contract-shaped dependencies (models, DTOs, records, enums).
`Uses`	Types the type uses without depending on them structurally.

Group	Meaning
`UnknownDependencies`	Names that could not be resolved to a type in the solution.

Every entry carries a confidence marker in square brackets:

```
<DEPENDENCIES>
Implements:
- IOrder [high]
UnknownDependencies:
- ICustomer [low]
- IRepository [low]
</DEPENDENCIES>
```

`[high]` marks a strong, architecture-relevant relation - typically a direct implementation, a constructor dependency or a central operational dependency. `[medium]` marks a relevant but less central relation, typically recurring usage. `[low]` marks a weak or data-oriented relation - typically DTO, model or enum usage, a contract-only transfer or an uncertain fallback. The markers are relative heuristics for the reader, not formal proof levels.

Which member types the lists name. The type of a property, a field and an event enters the type's dependency lists in the same way:

- Fields count at any accessibility, `static`, `readonly` or `const`; an enum constant names its enum. Compiler-generated backing fields are not counted.
- Events count in either form (field-like or with `add` / `remove` accessors), at any accessibility, `static`, in an interface or as an explicit interface implementation. The delegate type is listed unless its namespace is excluded; its type arguments and tuple elements are listed either way (`EventHandler<OrderPlacedArgs>` lists `OrderPlacedArgs`).
- For an array, a nullable, a generic argument or a tuple, the element type is listed.
- Keyword types are not listed: keywords such as `int` and `string`, and also `char`, `sbyte`, `ushort`, `uint`, `ulong`, `nint` and `nuint`; arrays of keyword types (`string[]`, `char[]`); nullable keywords (`int?`); and `dynamic`. An array is judged by its element, so a solution's own `Foo[]` stays. A type of the solution spelled like a keyword (a class `Char` or `Dynamic`) is listed as itself, and its `USED_BY` does not name the types that merely use the keyword, for example inside `List<char>`.

The referenced type's `USED_BY` names the declaring type in turn.

Note: `DEPENDENCIES` and `METHODS` are written for classes and interfaces even when they end up empty; the other sub-blocks are omitted when they have no content.

METHODS

`METHODS` contains one `METHOD` block per method, plus a comment when a type declares methods that are not rendered in this document - for example in a slice that projected the type without its methods. For a type the token budget reduced to its structure, that comment is followed by the signatures of the declared methods the document does not render as `METHOD` blocks (up to 80 lines, then a count of the rest), for example `<!-- 12 declared methods, signatures only: the budget took this type's source out (see the budget note) -->`. Depending on the detail settings, a method is written in one of the compression modes described below.

USED_BY

`USED_BY` lists the types (or methods, inside a method block) that use this element, one name per line:

```
<USED_BY>
- Acme.OrderService
- Acme.OrderController
</USED_BY>
```

This list is a statement about the document, not about the whole solution. It names only the callers that are rendered in this document; a caller outside the slice is absent, and when no caller is inside the slice the block is missing entirely. The lists are also resolved statically: reflection, dependency-injection containers, serializers and external assemblies stay invisible. **The absence of callers is therefore not proof of dead code.** To get the full caller list of a symbol, ask the producing tool for its caller axis.

Source-code blocks

`TYPE_CODE`, `METHOD_CODE` and `FILE_CODE` contain verbatim source code in a fenced block. The fence adapts to the content: when the code itself contains a run of backticks, the fence is made longer than that run, so the code block cannot be closed early by its own content. When a source file cannot be read or the code of an element cannot be extracted, the block contains a short comment saying so instead of the code.

Two opt-in annotations can be added to the tags:

- **Line numbers** (`lines="START-END"`): every `METHOD`, `CLASS`, `INTERFACE` and `ENUM` tag gets the 1-based line range of the declaration, in the same convention as the IDE status bar. When the feature is on but no line data is available - for example after reloading a stored analysis - the attribute reads `lines="n/a"` instead of being silently absent.
- **Quality metrics**: method tags get `complexity="N"` (cyclomatic complexity) and `loc="N"` (lines of code); type tags get `ce="N"` (efferent coupling), `ca="N"` (afferent coupling) and `members="N"` (member count). In a partial export - a slice, a task pack or a selection - `ca` is counted over the whole solution the export was cut from, so it is the same number a full export prints, not the number of referencers the slice happened to keep, and the `QUALITY_HOTSPOTS` coupling line prints the same value. A type nothing references reads `ca="0"`.

5.11 Method compression modes

The document writes methods in one of two compression modes. Which modes actually occur is stated in the `COMPRESSION_LEGEND` of the document itself.

Mode	Rendered as	Content
Full	`<METHOD>`	All sub-blocks: `METHOD_INFO`, `DEPENDENCIES`, `USED_BY`, `SEMANTICS`, `SUMMARY`, `AI_TAGS`, `METHOD_CODE`.
Compact	`<METHOD compact="isolated">`	A single line.

A missing `compact=` attribute means full mode. The compact line has this shape:

```
<METHOD compact="isolated">
ProcessOrder(int orderId) → Task<bool> | public static | Role: service
</METHOD>
```

It contains the signature, the return type (or `void`), the accessibility (or `private` when none is recorded), the declared modifiers and the resolved role. Because this line is the only information the reader gets about a compacted method, it carries the full modifier list - a dropped `override` or `async` would not be recoverable from anywhere else in the document.

The mode of a method is the result of several rules. A method is written compact only when its detail level is not `Full`, it does not belong to the primary or near flow, it is private, and it has no documentation summary, no AI annotations, no callers and no dependencies. Per-node overrides and the auto-compression heuristic can move any method into compact or detailed rendering.

`METHOD_INFO` inside a full method block contains the objective method facts: `Name`, `Signature`, `Return`, `Access`, `Modifiers`, `ReturnKind`, `Parameters`, and the flags `IsStatic`, `IsAsync`, `IsExtension`, `ThrowsDetected`, `UsesLinq` and `UsesReflection` (each written only when true). `UsesReflection` is set by any `typeof` expression or a bare reflection member name; it is not the same reading as the `reflection` trait of `find_by_code_traits`, and the document's SPEC block says so. A method block additionally carries `DEPENDENCIES` with a `Types:` list and a `Calls:` list of the methods it invokes in the solution, and a `USED_BY` list of its callers. The `Types:` list follows the rule of the type-level lists (see `DEPENDENCIES`): no keyword types, keyword arrays, nullable keywords or `dynamic`, while a solution's own type spelled like a keyword stays. A type used only as an element of a tuple inside a generic type (`new Box<(Order, int)>()`, `Action<(string, Change<int>)>`) is named there, and `find_usages` names the method as its user; a field typed as an array of tuples (`(Order, int)[]`) likewise counts the tuple's element types as used. The `Calls:` list includes the constructors the method calls: `new OrderService(repo)` appears as `OrderService.OrderService(IRepository)`. A constructor of an external type is not listed, and a non-public constructor is filtered like a non-public method.

5.12 The legends

PATH_LEGEND and alias rules

Long type names repeat throughout the graph sections, so the document abbreviates them. `PATH_LEGEND` is the decoding table and maps every abbreviated type name to its alias, sorted by name:

```
<PATH_LEGEND>
ColorBlendCalculator => CBC
ColorMixResult => CMR2
IColorMixService => ICMS
</PATH_LEGEND>
```

The alias rules are strict, and reading them correctly matters:

- Aliases may replace type names **only** in these ten sections: `DOMAIN`, `ARCHITECTURE_FLOW`, `INTERFACE_RELATIONS`, `ENTRY_POINTS`, `ENTRY_POINT_FLOW`, `SERVICE_DEPENDENCY_GRAPH`, `LAYER_MAP`, `CLASS_DEPENDENCY_GRAPH`, `METHOD_GRAPH`, `METHOD_USED_BY_GRAPH`. Everywhere else - code blocks included - every name is literal.
- `PATH_LEGEND` itself prints the mapping; it does not substitute.

- `ROLE_GRAPH` looks like the other graph sections but carries no type names: its edges connect roles (`Service -> Interface`), so it never uses the mapping.

Aliases are derived from the type name's capital letters. Names longer than twelve characters get an alias from their initials; a collision gets a numeric suffix, so two types with the same initials become `CA` and `CA2` . All remaining types are added alphabetically in a second pass, using their initials or, when there are too few, the first two characters of the name. Two rules keep the table unambiguous:

- A name shorter than two characters never gets an alias. The only alias it could carry is itself, which compresses nothing while still claiming that a bare letter refers to that type.
- Names of twelve characters or fewer are reserved first, so a longer type's alias cannot shadow them. They can still receive their own alias in the second pass (initials, or the first two characters).

The mapping is keyed by the simple type name. Two types with the same simple name in different namespaces therefore share one alias; the per-type `[Layer: ...]` and `[Role: ...]` fields are the unambiguous source when this matters.

If layout rendering is switched off, no aliases are built, `PATH_LEGEND` and the alias rules are omitted, and every name is written out in full.

COMPRESSION_LEGEND

`COMPRESSION_LEGEND` is always rendered. It tells the reader

- which compression modes the document actually uses (one or both),
- how a method is selected for full or compact rendering,
- what a missing `compact=` attribute means,
- the provenance vocabulary of the `SEMANTICS` blocks (`Resolved` , the breakdown form and the four tokens),
- and the caveat that applies to every caller list: the lists name only the callers this document renders, and they are statically resolved. Reflection, dependency-injection containers, serializers and external assemblies stay invisible; the absence of callers is not proof of dead code.

Read this section first when you interpret a rendered document - it describes exactly the notation in front of you.

5.13 Detail levels and auto-compression

How much of a type or method is written is decided by a **detail level** with four steps:

Level	Effect
`Compact`	Shortest form. For a method, the single compact line.
`Normal`	Standard form: type information, semantics, summary, structure, dependencies, methods.
`Detailed`	Normal plus the extended fields.
`Source`	Detailed plus the verbatim source-code block.

A context template holds one detail preset per level (`Compact` , `Normal` , `Detailed` , `Source`), and a node in the solution tree can override its level individually. Without a per-node setting, the `Normal` preset governs the whole export.

The **auto-compression heuristic** proposes a level per method so that an export does not have to be tuned by hand. It runs on first use of a solution when enabled and can also be triggered manually:

Rule	Result
AI priority `high` or `important`	`Detailed`
AI priority `low` or `nice to have`	`Compact`
Entry point (static `Main`, or a public method of a `Controller` type)	`Detailed`
Public method with at least 3 callers	`Normal`
Everything else	`Compact`

The caller threshold, the three force rules and the master switch are configurable in the active compression rules profile. The heuristic only proposes a level; the context template turns that level into a concrete detail preset, so changing the template changes the final rendering while the heuristic stays the same.

Two more axes influence the result:

- **Expansion strategies** decide how far the neighborhood of the selection is expanded before rendering - only the explicitly selected nodes, their direct neighbors, or everything reachable within the configured depth. They are separate from the detail level: expansion decides *what* enters the document, the detail level decides *how much* of it is written.
- **The method graph mode** on a detail preset decides how much of a method signature the method graph carries (four values: complete graph, semantic edges only, with signature, with body). The built-in presets mostly use the signature variant.

5.14 Token budget and trimming

A document can be limited to a token budget. The budget and its trimming behavior come from the active **pipeline profile**:

Setting	Default	Meaning
`BudgetedTrimmingEnabled`	off	Master switch for the budget-trimming pipeline.
`MaxTokenBudget`	60 000	The token budget for the rendered document. Lower values are raised to a floor of 8 000.
`MaxOvershootPercent`	50	How far the document may exceed the budget before types are dropped. The admissible ceiling is `budget × (1 + percent / 100)`.
`AutoBypassSnapshots`	on	Skips trimming in read-only snapshot tabs.
`ShowBudgetHeaderInOutput`	off	Writes the budget comment into the document (see below).

Three built-in pipeline profiles ship with the product:

Profile	Behavior
`Default`	Trimming off, snapshots bypassed. Recommended default.

Profile	Behavior
`Aggressive Trimming`	Trimming on with a 20 000 token budget, a +10 % overshoot tolerance and the budget header enabled.
`No Trimming (Snapshot-Mode)`	Trimming off with a 200 000 token budget and no automatic bypass. Suited to models with a very large context window and to reproducible comparisons.

In the MCP render path the effective budget is resolved from the first of these tiers that is set:

```
explicit call parameter  >  MCP profile  >  context template  >  global pipeline profile
```

The MCP `budget` parameter of the slice tools (`get_context` , `explain_symbol` , `pack_for_task` , `prepare_task`) is the explicit tier and is raised to 8 000 when lower. When no budget is set at all, the slice tools still render against a ceiling (the active MCP profile's budget, otherwise about 10 000 tokens), so a slice stays bounded.

Note: a token budget set on a context template takes effect only in the MCP render path. The GUI and CLI renders ignore it, so that stored snapshots stay reproducible.

What trimming does

Trimming runs in two stages, both aimed at the same number:

1. **Type pruning.** Over-budget selected types are first reduced to their structure - they keep their identity and members, and their block lists the signatures of the methods they declare, but they lose their source code - to reach the target. That signature list is charged against the budget too. Only if that is not enough are types dropped, down to the ceiling. At least one type always survives, and the export pruner never takes the code of the focal, top-ranked type away. Retrieval slices (`get_context` , `explain_symbol` , `pack_for_task` , `prepare_task`) may still degrade the focal type to structure, with a warning and budget recommendation, when its source alone exceeds the slice budget.
2. **Method trimming.** The remaining methods are compressed along the compression ladder until the estimate fits.

The trimming is estimate-based, so the export path measures the finished document and corrects itself in bounded passes. A correction pass may lower the internal floor from 8 000 to 2 000 tokens so that a tight target is reachable.

The compression ladder

The trimming pipeline orders methods by importance and walks them down a five-step ladder until the estimated size fits the budget:

```
Drop = 0, NameOnly = 1, SignatureOnly = 2, SignatureDoc = 3, FullBody = 4
```

Each method starts on a rung derived from its detail level (`Source` and `Detailed` start at full body, `Normal` at signature plus documentation, `Compact` at signature only). The methods with the lowest score are moved down first.

Note: the ladder decides the order and the number of removals, not the rendering of the survivors. Only the `Drop` rung removes a method from the output; whether a surviving method is written with its body is decided by the detail preset, independently of the ladder. Mapping the intermediate rungs onto the render paths is a planned

product decision, not current behavior.

Two settings of the compression rules profile are not yet in effect: the trimming mode `Sweep` (which is defined but behaves like the default `Greedy` mode) and the tiebreaker order list (the comparator uses a fixed order of score, lines of code, name and qualified key).

The budget header

When `ShowBudgetHeaderInOutput` is enabled on the active pipeline profile, the document starts with an HTML comment that reports the trimming result:

```
<!-- token_budget: 60000 | est_after_trim: 58423 | trimmed: 142 methods (6 dropped) -->
```

A run that bypassed the trimming prints one of these instead:

```
<!-- token_budget: 60000 | bypassed (read-only snapshot) -->
<!-- token_budget: 60000 | bypassed (trimming disabled) -->
```

When a token budget is enforced, the render first removes or reduces whole types to fit it, and both forms then end in a fourth part:

```
<!-- token_budget: 8000 | est_after_trim: 7900 | trimmed: 12 methods (0 dropped) | types: 180
dropped, 3 to structure -->
```

A bypassed line carries the same part, because types are cut whether or not method trimming runs.

Read the fields carefully:

- `token_budget` is the budget **you** asked for. It is not necessarily the number the trim aimed at internally, but it is always the user-facing budget.
- `est_after_trim` is the estimated size after trimming. It may legitimately exceed the budget when overshoot was granted, and it is **not** the delivered size: it states what the document would cost if the render honored every compression level. Because only the `Drop` level reaches the output, the real document is usually larger.
- `trimmed: N methods` counts the methods that ended below their starting level. The dropped methods are included in that number, so `dropped <= trimmed <= total`.
- `types: N dropped, M to structure` counts the types the budget removed from the document and the selected types it reduced to structure (signatures, no bodies) and kept. A type that was first reduced and then removed counts as dropped only. The part is absent when no budget was enforced.

The header is off by default, and only the `Aggressive Trimming` built-in profile enables it.

5.15 Document modes

Four document modes decide which sections exist at all. You do not select them directly - each render path uses its own - but they explain why two documents of the same solution can differ.

Mode	Used by	Effect
Full export	`export_markdown`, the GUI export, the CLI export	Nothing is removed.
Lean	`get_context`, `explain_symbol`, `pack_for_task`, `prepare_task`, `architecture_overview` (default)	Drops the format preamble (`SPEC` and `META`) and every enabled graph section that would only contain `- none`. `PATH_LEGEND`, `COMPRESSION_LEGEND` and all non-empty sections stay.
Structure-only	`architecture_overview`	Drops the sections with source code and type trees (`SOLUTION`, `FILE_INDEX`, `ENUM_SUMMARY`, `BUILD_AND_UI_FILES`, `AUXILIARY_FILES`) and keeps the macro sections, the legends and the domain summary. Quality hotspots are enabled in this mode.
Focused slice	`explain_symbol`, `get_context`, `pack_for_task`	Additionally drops the two solution-wide verbatim file sections `BUILD_AND_UI_FILES` and `AUXILIARY_FILES`.

The lean mode is the default for all slice tools; pass `lean: false` to get the full document with the format contract. `prepare_task` ignores the flag on its template-based render path, which the active MCP profile controls.

The bounded **architecture overview** caps each macro section except `LAYER_MAP` at 80 entries and appends a truncation note when the cap bites. `LAYER_MAP` lists every type up to 2,000 types; above that it shows only the per-layer counts, never a sample, and the leading `<TRUNCATION>` block names it as `LAYER_MAP: 0 of N`. For the complete type-to-layer map of such a solution use `export_markdown`, which does not apply this bound. The overview's `summaryOnly` mode collapses the other macro sections to the first five entries plus a count line, while `LAYER_MAP` becomes an aggregate with a total and per-layer counts. In that mode the document also renders plain type names instead of aliases, so it never advertises a legend it does not carry.

Four sections cannot be switched off through an MD profile because they have no profile slot: `COMPRESSION_LEGEND`, `BUILD_AND_UI_FILES`, `AUXILIARY_FILES` and `QUALITY_HOTSPOTS`. The first three are always on and relevance-filtered; quality hotspots are controlled by the quality-metrics setting.

5.16 What the document does not tell you

A few properties of the format are easy to misread. They are listed here so you can interpret a document correctly:

- **Caller lists are partial by construction.** `USED_BY` names only the callers rendered in the same document - also after a token budget removed types from it. In a slice, a caller outside the slice is missing and the block may be absent altogether.
- **Caller lists are static.** Reflection, dependency-injection containers, serializers and external assemblies are not visible to the analysis. A missing caller is not dead code.
- `inferred` **values are guesses.** They are useful hints, but they are not facts. Rely on `fact` and `ai` values when correctness matters.
- `lines="n/a"` **means "not measured here"**, not "zero" and not "the feature is off". `ca` is always measured, over the whole solution, even in a slice.
- **Empty fields and empty blocks are omitted.** A flag such as `IsStatic: true` appears only when the flag is set, so an absent flag means false. A missing optional field means it has no value in this render - not that it was empty in the source.
- `UnknownDependencies` **names what could not be resolved**, not what does not exist. It typically contains dependencies from external assemblies.

- **The namespace part of `QUALITY_HOTSPOTS` is omitted in partial exports**, and the document says why: measuring namespace coupling over a pruned view would understate the afferent and overstate the efferent coupling and could produce a misleadingly clean result.
- `QUALITY_FINDINGS` **in a partial document lists only the findings about the types it renders.** They are judged over the whole solution, so a type is not reported as unused because its users were left out; findings about types the document does not render are not listed.

5.17 A complete example

The following excerpt is real product output, abbreviated to the essentials. It shows a small solution with one project, one file and one class:

```
---
type: ai-builder-context
title: Acme.Minimal
description: Roslyn-generated AI-Builder-MD context for the Acme.Minimal solution.
---
<AI_CONTEXT_SPEC>
Purpose: Structured semantic representation of a .NET solution optimized for LLM understanding.
...
AliasRules:
5. PATH_LEGEND maps type names to short aliases.
6. Aliases may replace type names only in these sections: DOMAIN, ARCHITECTURE_FLOW, ...
7. PATH_LEGEND always contains the full type name mapping.
...
Sections:
- SPEC
- META
- PATH_LEGEND
- COMPRESSION_LEGEND
- DOMAIN
...
</AI_CONTEXT_SPEC>

<AI_CONTEXT_META>
Generator: AIContextBuilder
Version: 1.0
Format: AIContextMarkdown
Solution: Acme.Minimal
GeneratedAt: 2026-09-21T09:15:42Z
AnalyzerVersion: 0.5.464.xxx
Projects: 1
Files: 1
Types: 1
Classes: 1
Interfaces: 0
Enums: 0
Methods: 0
Properties: 0
Constructors: 0
DistinctDependencies: 0
</AI_CONTEXT_META>

<PATH_LEGEND>
Hello => HE
</PATH_LEGEND>

<COMPRESSION_LEGEND>
Format: Methods are emitted in one of two compression modes.
```

```
...
</COMPRESSION_LEGEND>

<DOMAIN>
Name: Acme.Minimal
CoreComponents:
</DOMAIN>

<ARCHITECTURE_FLOW>
Flows:
- none
</ARCHITECTURE_FLOW>

<FILE_INDEX>
[Project: Acme.Minimal]
**RelativePath:** src/Hello.cs
Types: 1
Methods: 0
</FILE_INDEX>

<SOLUTION>
Name: Acme.Minimal
Projects: 1
<PROJECT>
Name: Acme.Minimal
<FILE>
Name: Hello.cs
Path: src/Hello.cs
<CLASS>
**TypeName:** Hello
<STRUCTURE>
Access: public
</STRUCTURE>
<DEPENDENCIES>
</DEPENDENCIES>
<METHODS>
</METHODS>
</CLASS>
</FILE>
</PROJECT>
</SOLUTION>
```

The same content in the YAML notation begins like this:

```
type: ai-builder-context
title: Acme.Minimal
description: Roslyn-generated AI-Builder-MD context for the Acme.Minimal solution.
aiContextSpec: |
  Purpose: Structured semantic representation of a .NET solution optimized for LLM understanding.
  ...
aiContextMeta:
  generator: AIContextBuilder
  version: "1.0"
  format: AIContextMarkdown
  solution: Acme.Minimal
solution:
  name: Acme.Minimal
  projects:
  - name: Acme.Minimal
    files:
    - name: Hello.cs
      path: src/Hello.cs
      classes:
      - typeName: Hello
        structure:
          access: public
```

Both notations carry the same information: the frontmatter fields as top-level keys, the sections as mappings, repeated blocks as sequences, and the format contract as one string field.

6 Templates, rendering and markup analysis

A template is the configuration root of an export. It decides which prompt is used, which sections the Markdown document contains, how much detail each symbol gets, how far the selection walk expands, and which profiles govern scoring, token budget and quality findings. This chapter describes the template model, the built-in catalogs, how the document is assembled, and how AICB analyzes XAML and AXAML markup - the part of a solution that the C# symbol graph cannot see.

6.1 What a template controls

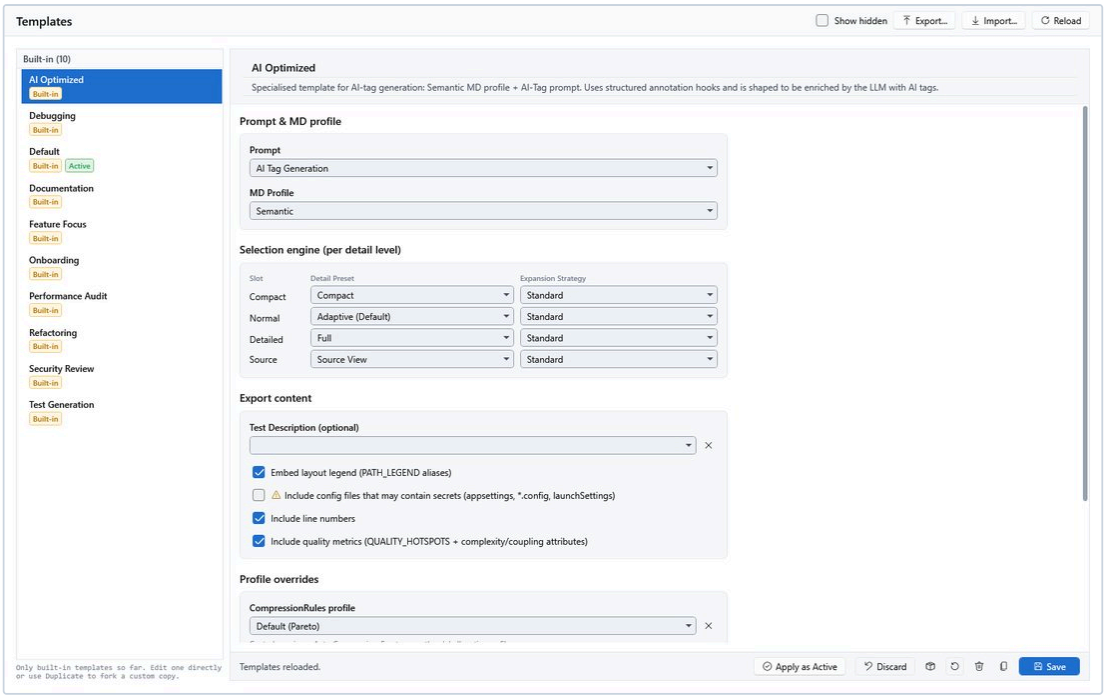
A template is a master entity: it has a name, an optional description, and a set of references to other master entities plus its own value settings. You maintain templates under Configuration → Templates.

Slot	Refers to	Required	What it decides
`Prompt`	Prompt	yes	The prompt sent to the LLM when this template runs. The Context Builder can override it per run.
`MD Profile`	MD profile	yes	Which sections the Markdown document contains and which tag schema each of them uses.
`Compact`, `Normal`, `Detailed`, `Source` (Detail Preset)	Detail preset	yes	How much of each symbol is rendered at that detail level.
`Compact`, `Normal`, `Detailed`, `Source` (Expansion Strategy)	Expansion strategy	optional	How far the selection walk expands at that detail level. When all four slots are empty, the template's single legacy expansion reference applies. Used by the GUI tree only; the MCP server and the CLI do not read these slots.
`CompressionRules profile`	Compression rules	optional	Scoring multipliers and the auto-compression heuristic. Empty = use the globally active profile.
`Pipeline profile`	Pipeline profile	optional	Token-budget pipeline settings and snapshot auto-reload. Empty = use the globally active profile.
`Token budget (MCP render)`	-	optional	Maximum output tokens when the MCP server renders through this template. Floor 8,000. Affects the MCP render path only - GUI and CLI exports are unaffected. When set, it forces trimming; a token budget set on the MCP profile wins over it.
`Expansion around the MCP start point`	Expansion strategy	optional	How far a `prepare_task` bundle grows around the symbols the task names when the MCP server or the CLI renders through this template. Empty = the MCP's own default. The GUI tree does not read it, and a whole-solution export is not affected. The CLI reads a change on its next run; an MCP server that is already running reads it only after a restart.
`Insights profile`	Quality profile	optional	Producer toggles and thresholds for quality findings. Empty = use the globally active profile.
`Test Description (optional)`	Test description	optional	A benchmark/test description attached to runs.

The template's own switches:

Setting (label in the editor)	Default	Effect
`Embed layout legend (PATH_LEGEND aliases)`	on	Applies the compact type-alias presentation to the graphs and several other sections, and emits the `PATH_LEGEND` decoding table. Off renders plain type names and drops the legend. Off suits refactoring-style runs where the layer map is the dominant signal and aliases add overhead.
`Include config files that may contain secrets (appsettings, *.config, launchSettings)`	off	Emits `appsettings.json`, `appsettings.*.json`, `*.config` (web/app/nuget.config) and `launchSettings.json` verbatim in `AUXILIARY_FILES`, including any connection strings and API keys. Only enable this for a solution whose configuration holds no secrets, or review the output before sharing it.
`Include line numbers`	off	Adds a `lines="START-END"` attribute (1-based) to every method and type tag. Useful when the model must point at source locations or when line spans are compared across runs.
`Include quality metrics (QUALITY_HOTSPOTS + complexity/coupling attributes)`	off	Adds the `QUALITY_HOTSPOTS` section and a `complexity=""` / `ce=""` attribute to method and type tags.

An empty profile slot means "use the globally active profile". The global defaults are chosen with `Apply as Active` in the matching Settings panel. For the template itself, the MCP render path resolves: the facet slot of the active MCP profile, then the profile's profile-wide template, then the active context template.



The Templates page: a context template with its prompt, MD profile, detail-preset slots and profile overrides

Working with master entries:

- Saving an edited built-in marks it as overridden; `Restore Built-In` resets it to its code defaults and clears that flag.

- `Duplicate` creates a custom copy - the way to fork a built-in.
- `Delete` hides a built-in (recoverable via `Restore Built-In`); a custom entry is removed permanently.
- `Export as Built-In` shows the entry as a C# snippet; it is intended for the product's own development and has no effect on your installation.
- A custom entry that other entities reference shows an `IN USE` pill.

6.2 Built-in templates

Ten templates ship with the product.

Template	Purpose	MD profile	Detail slots (Compact / Normal / Detailed / Source)	Expansion	Notable settings
`Default`	Balanced standard context generation; a good starting point when no specific task is in mind.	Full	Compact / Adaptive / Full / Source View	Standard	None - it states "use the global settings".
`Refactoring`	Architecture-focused work: CQRS migrations, SOLID review, layering audits.	Full	Compact / Adaptive / Full / Source View	Standard	Layout legend off; quality metrics on; Insights profile `Strict`.
`Debugging`	Bug hunting with full detail and deep call-chain expansion.	Full	Adaptive / Full / Full / Source View	Deep	Line numbers on; Compression rules `Lenient`.
`Feature Focus`	Targeted feature extensions with a minimal token footprint.	Compact	Compact / Compact / Adaptive / Source View	Direct-only	Compression rules `Strict`; Pipeline profile `Aggressive Trimming`.
`AI Optimized`	AI-tag generation: structured annotation hooks meant to be enriched by the model.	Semantic	Compact / Adaptive / Full / Source View	Standard	Line numbers and quality metrics on; pins the built-in Default compression, pipeline and insights profiles. Also the profile-wide template of the Default MCP profile.
`Onboarding`	Public API surface for code-reading tours and external integration review.	API Surface Only	Ultra-Compact / Adaptive / Full / Full	Standard	Insights profile `Disabled`.
`Test Generation`	Generates xUnit + FluentAssertions tests for the selected classes.	Compact	Compact / Source View / Source View / Source View	Direct-only	None; the complex-untested producer stays inherited because it targets test generation.

Template	Purpose	MD profile	Detail slots (Compact / Normal / Detailed / Source)	Expansion	Notable settings
`Documentation`	Produces XmlDoc comments for the public API surface.	API Surface Only	Ultra-Compact / Adaptive / Full / Full	Direct-only	Insights profile `Disabled`.
`Performance Audit`	Hot-path identification, allocation hotspots, throughput-critical paths.	Diagnostics	Compact / Adaptive / Full / Source View	Standard	Line numbers and quality metrics on; Insights profile `Strict`.
`Security Review`	Input-validation gaps in public APIs, validation-bypass paths, threat hypotheses.	Full	Compact / Adaptive / Full / Source View	Standard	Line numbers and quality metrics on; Compression rules `Lenient`; Insights profile `Strict`.

Note: `Onboarding` and `Documentation` use the `Full` preset in the Source slot instead of `Source View`, because the `API Surface Only` MD profile does not offer the Source level.

6.3 Detail presets

A detail preset controls how much of a symbol is written at one detail level. It is the axis for method-level reduction; the tag schemas (below) decide the line shape.

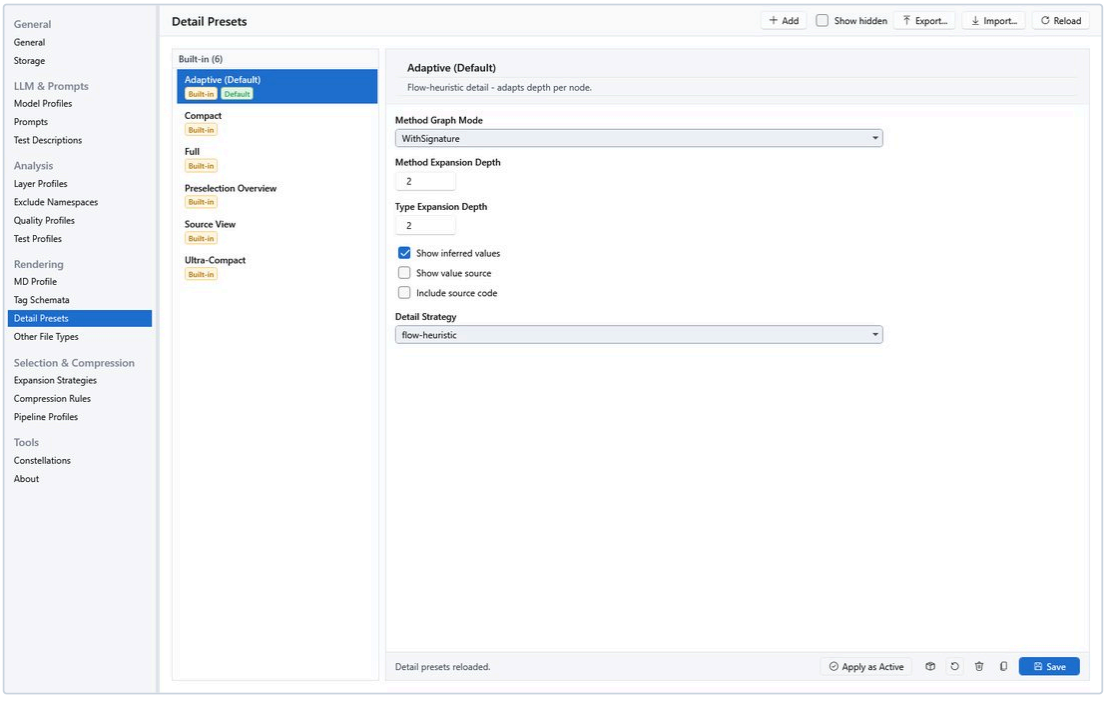
Field (label in the editor)	Default for a new preset	What it does
`Method Graph Mode`	`WithSignature`	Render mode for method call graphs: `Full`, `SemanticOnly`, `WithSignature`, `WithBody`.
`Method Expansion Depth`	2	How many call-graph hops the walk expands from a node. 1 = direct neighbors only, 2-3 = small clusters, higher = larger context. Used when the active expansion strategy works in reachable mode.
`Type Expansion Depth`	2	How many type-dependency hops (uses / used-by) to expand - the breadth of type context around a selection.
`Show inferred values`	on	Include inferred / resolved values in the output, for example constant expressions and default parameter values.
`Show value source`	off	Annotate inferred values with where they came from, for traceability.
`Include source code`	off	Append a source-code block for code nodes that resolve to this preset. Typically on for the preset in the template's `Source` slot.
`Detail Strategy`	`flow-heuristic`	The reduction behavior. Only three values can be selected: `aggressive` = compact, no adaptive; `flow-heuristic` = compact where the flow heuristic allows it for the node; `none` = full detail.

The six built-in presets:

Preset	Method graph mode	Method depth	Type depth	Inferred values	Value source	Strategy	Source code
`Full`	`WithSignature`	5	5	yes	yes	`none`	-
`Adaptive (Default)`	`WithSignature`	2	2	yes	no	`flow-heuristic`	-
`Compact`	`SemanticOnly`	1	1	no	no	`aggressive`	-
`Source View`	`WithSignature`	5	5	yes	yes	`none`	yes
`Preselection Overview`	`WithSignature`	0	0	no	no	`aggressive`	-
`Ultra-Compact`	`SemanticOnly`	0	0	no	no	`aggressive`	-

`Adaptive (Default)` carries the default marker; `Source View` is configured like `Full` plus `Include source code`; `Preselection Overview` (class list and signatures, no bodies) and `Ultra-Compact` (type names and tags only) are the two most token-efficient presets. Note: the two-stage preselection run type is not yet released; the `Preselection Overview` preset itself is selectable like any other.

How the slots reach the document: the solution tree resolves a detail level per node, and the template maps each level to its preset. An export that does not build a per-node selection - for example the whole-solution export - uses the `Normal` preset for everything.



The Detail Presets page

6.4 MD profiles

An MD profile maps every section to a tag schema. It has exactly one slot per section type - 27 slots. The four code-structure slots (`Class` , `Method` , `Interface` , `Enum`) are staged; the other 23 are single.

A staged slot holds a schema per level plus two settings:

Setting	Meaning
`Compact` / `Normal` / `Detailed`	The tag schema used when a node resolves to that level. The Source level renders the `Detailed` schema as well.
`Default level:`	The level that applies when neither a node override nor a session override applies.
`Allow Source`	Marks the slot as able to serve the Source level; nodes of this type can then be switched to Source in the solution tree.

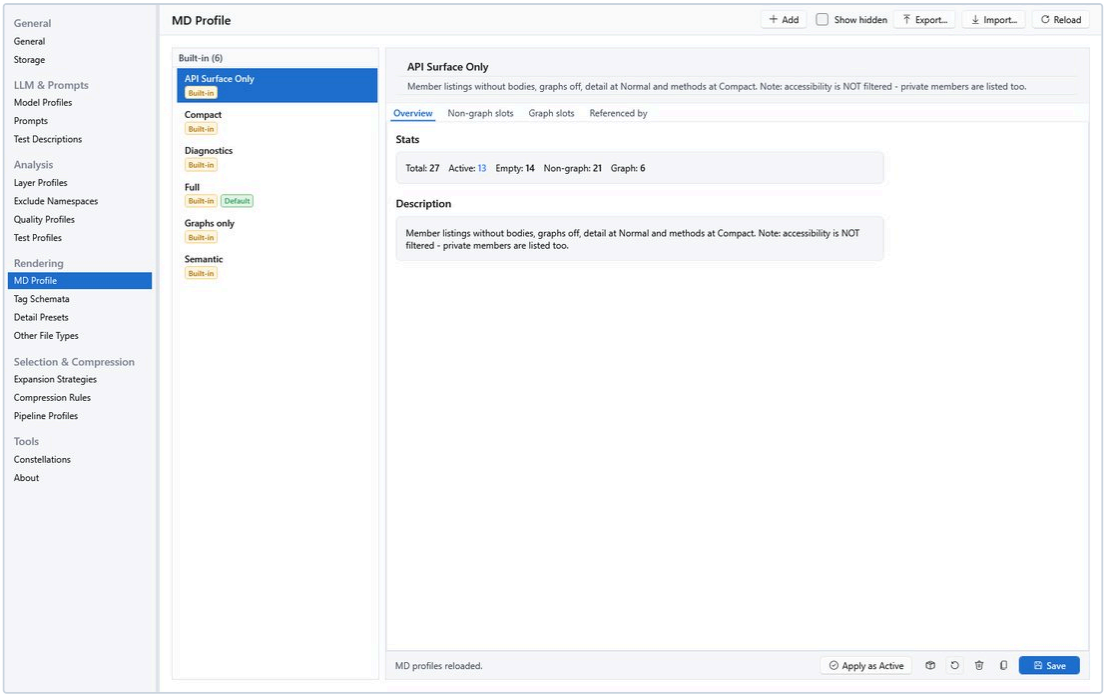
A single slot holds one schema. An empty single slot switches its section off. A staged slot with an empty rung does not switch anything off: it falls back to the built-in sub-block schema for that level. Graph slots are a special case - see below.

The six built-in profiles, with the descriptions you see in the product:

Profile	Description	What it leaves in the document
`Full`	"Every slot filled with standard tag schemas. Source level allowed."	Everything the profile can control; code structure at Detailed. This is the initial global default.
`Semantic`	"Only semantically important slots filled. Detail slots at Normal level (Compact/Detailed empty → cascade)."	The semantic sections and code structure at Normal; `ENUM_SUMMARY`, `FILE_INDEX`, `PATH_LEGEND` and all graphs off.
`Compact`	"Minimal slot occupancy. Detail slots at Compact level. Graph slots empty."	Spec, Meta, AI prompt and the structure sections at Compact; most analysis sections and all graphs off.
`Graphs only`	"The 6 graph slots plus the Meta header; every other SLOTTED section off. Code structure and the slot-less sections (legend, build/UI + auxiliary files) still render; code structure at the Compact level its detail slots declare."	The six graph sections and the Meta header; code structure at Compact.
`API Surface Only`	"Member listings without bodies, graphs off, detail at Normal and methods at Compact. Note: accessibility is NOT filtered - private members are listed too."	Member listings without bodies; all graphs off.
`Diagnostics`	"Method info + dependencies + used-by + perf flags, no code. Graphs for architecture visualization. Performance and security audit profile."	Method info, dependencies, used-by and performance flags; five graphs; the role graph stays off.

Note: `API Surface Only` does not filter by accessibility - private members are listed like any other.

The profile editor has four tabs: `Overview` (slot statistics: Total, Active, Empty, Non-graph, Graph, plus the description), `Non-graph slots` (the staged and single section slots), `Graph slots`, and `Referenced by` (which templates use this profile). Graph slots carry the hint that layout and compression are controlled globally under Settings → General → `Default graph layout (compression)` with a per-tab override in the Context Builder.



The MD Profile page: slot statistics and the staged and single slots

Graph slots and graph selection

The renderer knows ten graphs. Six of them have an MD-profile slot:

Graph section	Profile slot
`ARCHITECTURE_FLOW`	no
`INTERFACE_RELATIONS`	no
`ENTRY_POINTS`	no
`ENTRY_POINT_FLOW`	no
`SERVICE_DEPENDENCY_GRAPH`	yes
`LAYER_MAP`	yes
`CLASS_DEPENDENCY_GRAPH`	yes
`ROLE_GRAPH`	yes
`METHOD_GRAPH`	yes
`METHOD_USED_BY_GRAPH`	yes

The four macro graphs have no slot: in renders without a Graph Selection (CLI, MCP) they are always included, and in the Context Builder you tick them in the `Graph Selection` panel. The six slotted graphs are seeded from the active profile - a slot that carries a schema means the graph is on by default. The checkboxes are yours to change per tab, so an empty graph slot in a profile does not force the graph off in the GUI; in renders that build their graph set from the profile (CLI, MCP), an empty slot leaves that graph out.

Sections that no MD profile controls

Five rendered sections have no slot at all and cannot be switched off through a profile:

- `COMPRESSION_LEGEND` - always on.
- `BUILD_AND_UI_FILES` - always on, relevance-filtered.
- `AUXILIARY_FILES` - always on, relevance-filtered; the sensitive-config subset is the one opt-in (see the template switch above).
- `QUALITY_HOTSPOTS` - gated by the template's `Include quality metrics` switch.
- `QUALITY_FINDINGS` - rendered when the analysis produced findings.

6.5 Tag schemas

A tag schema decides which fields a block contains, in which order, and how each field is written. It belongs to exactly one schema type. There are 52 built-in schemas; per schema type exactly one carries the default marker. That default is used when no profile or slot resolves. An empty single-schema slot suppresses its section; an empty rung in a staged slot falls back to the built-in sub-block schema.

A schema is a list of fields. Each field row has:

Column (label in the editor)	Meaning
Field name	The label written into the document, for example `TypeName` or `Layer`.
Source	The mapping onto analysis data. `facts.*` are deterministic Roslyn facts, `resolved.*` are values from the resolution chain; only paths from the built-in list are accepted, and the list offers only paths that produce output for the schema type. A field whose stored source is no longer offered shows a warning, and the schema cannot be saved until the field is changed or removed.
`Active`	When unchecked, the field is kept in the schema but skipped during rendering.

The field's render strategy decides the shape of the emitted line. The renderer provides twelve strategies; the built-in schemas use ten of them. `json` and `yaml` are available to custom fields but are not used by built-in schema fields:

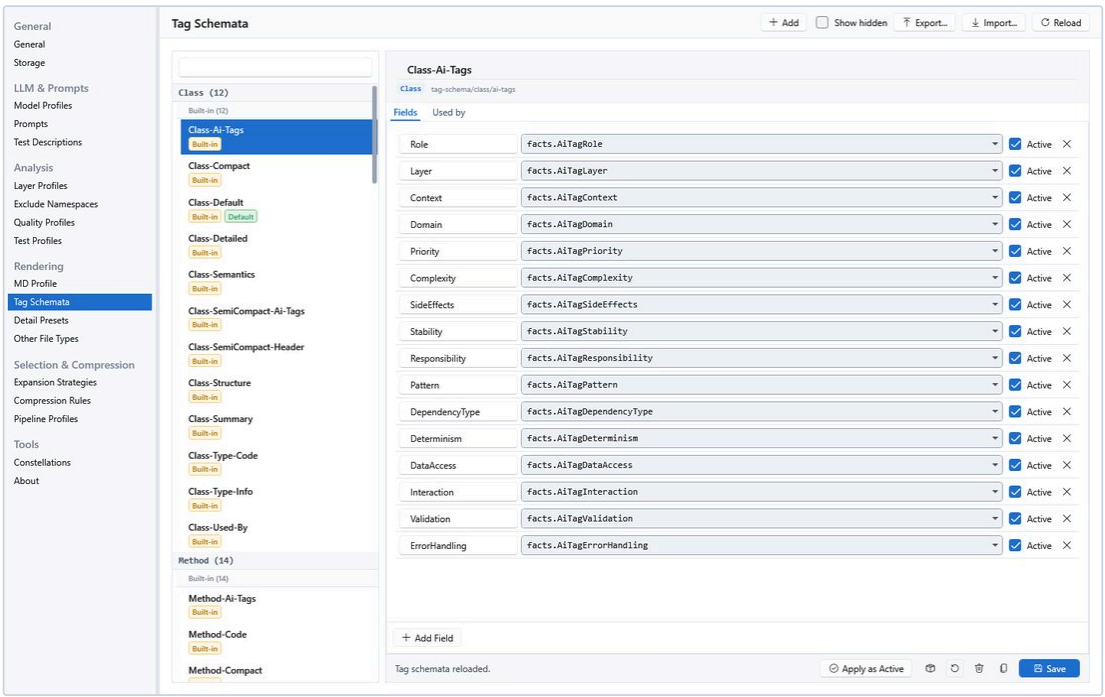
Strategy	Output form
`strong`	`**Field:** value` (bold label).
`plain`	`Field: value`. The fallback for an unknown or empty strategy.
`tag`	`[Field: value]`.
`quote`	`> Field: value`.
`list`	`Field:` followed by one `- item` line per value; a single scalar value renders as one bullet line, `- Field: value`.
`compact-list`	`Field: a, b, c` for multiple values; a single value behaves like `plain`.
`raw`	The value only - no label, no trimming. For fields whose surrounding section already provides the wrap.
`bullet-list`	`- item` lines without a field header, for sub-blocks such as `USED_BY`.

Strategy	Output form
`flag`	Boolean flags: the field appears only when true.
`llm-natural-md`	Natural GFM for multi-line fields: a `### Field` heading plus a fenced code block with `csharp` / `xml` auto-detection.
`json`	A JSON object member, for machine-readable pipelines.
`yaml`	A YAML mapping entry; multi-line strings use the `|-` block literal.

The 24 schema types an MD profile slots, and how:

Group	Schema types	Slot mode
Code structure	`Class` , `Method` , `Interface` , `Enum`	staged
Section	`Spec` , `Meta` , `AiPrompt` , `Domain` , `ArchitectureFlow` , `InterfaceRelations` , `EntryPoints` , `EntryPointFlow` , `EnumSummary` , `FileIndex` , `PathLegend` , `Solution` , `Project` , `File`	single
Graph	`MethodGraph` , `MethodUsedByGraph` , `ServiceDependencyGraph` , `LayerMap` , `ClassDependencyGraph` , `RoleGraph`	single

The schema editor has three tabs: `Fields` (name, source, active, add and remove), `Layout` , and `Used by` (which MD profiles reference this schema).



The Tag Schemata page: the fields of a schema with their sources

Note: the `Layout` tab is reserved for future renderer wiring. Its four values - `Node label` (default `MethodSignature + DeclaringType`), `Edge style` (default `Arrow with caller→callee label`), `Clustering` (default `By layer`) and `Depth limit (1-10)` (default 3) - are persisted with the schema but are not consumed by any renderer yet; the

fields are shown read-only. App-global compression on/off lives under Settings → General.

A schema you create yourself reaches the export when an MD profile's `Class`, `Interface`, `Enum` or `Method` row, or the `Sections` panel of a tab, picks it: it becomes the first lines of every such block (for a method, its `METHOD_INFO`) and fills every field it offers where the code has the value. A copy of a built-in block replaces just that block. The `SOLUTION`, `PROJECT` and `FILE` header lines render from their schemas; the built-in ones give the usual header. Under each schema the editor says where it shows up in an export, or why it does not.

6.6 Expansion strategies

An expansion strategy controls how the selection walk follows the graph from the selected nodes: the mode, the depth, eight inclusion flags and four stop conditions. You maintain them under Settings → Selection & Compression → `Expansion Strategies`.

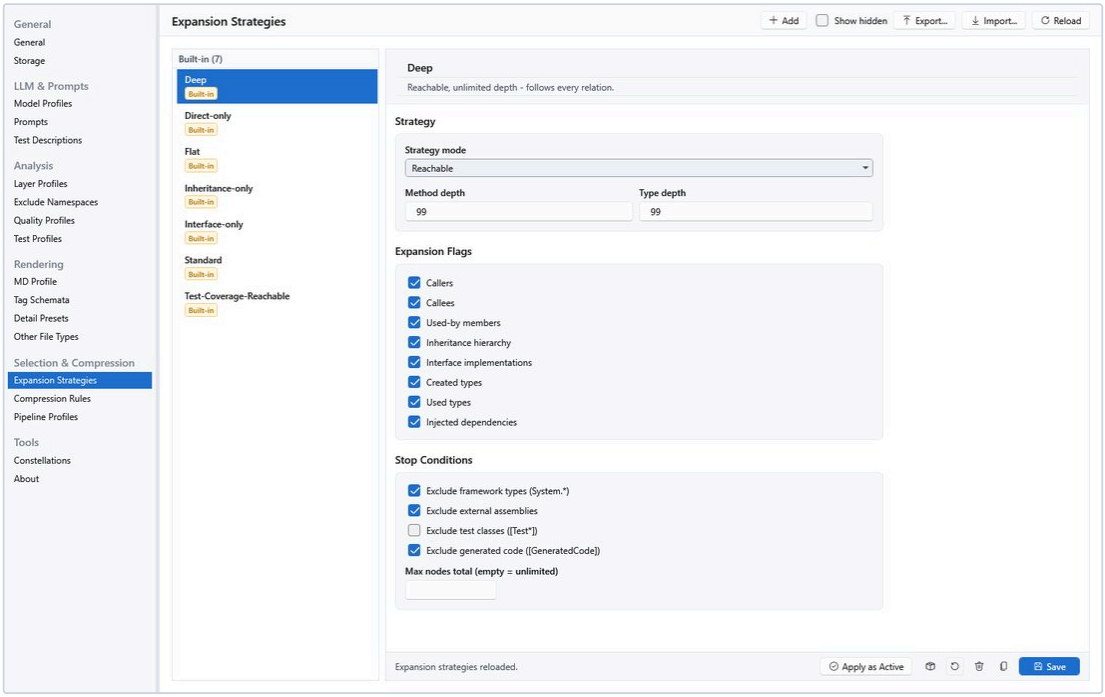
Editor fields: `Strategy mode`, `Method depth`, `Type depth`; the flags `Callers`, `Callees`, `Used-by members`, `Inheritance hierarchy`, `Interface implementations`, `Created types`, `Used types`, `Injected dependencies`; the stop conditions `Exclude framework types (System.*)`, `Exclude external assemblies`, `Exclude test classes ([Test*])`, `Exclude generated code ([GeneratedCode])` and `Max nodes total (empty = unlimited)`.

The seven built-ins:

Strategy	Mode	Method depth	Type depth	What it walks	Notable
`Deep`	Reachable	99	99	Every relation, unlimited depth.	Includes test classes.
`Standard`	Select	2	1	Seeds only - no expansion. The neutral starting point.	Depths and flags are reserved for a custom copy promoted to Reachable.
`Flat`	Frontier	1	1	One-hop frontier: seeds plus their direct injected dependencies.	
`Direct-only`	Frontier	1	0	Direct callers and callees, one hop.	
`Inheritance-only`	Reachable	0	5	Only the inheritance hierarchy.	Use for class hierarchies, OO audits, Liskov checks.
`Interface-only`	Reachable	0	5	Only interface implementations.	Use for "all `IAuthenticator` implementations", polymorphism mapping.
`Test-Coverage-Reachable`	Reachable	10	10	Everything reachable from test classes - what tests actually touch.	Test classes stay as roots in the walk.

All built-ins except `Deep` and `Test-Coverage-Reachable` stop at test classes, and all of them stop at framework types, external assemblies and generated code.

The `Interface implementations` flag follows an interface to the types that implement it (and a class to the interfaces it implements), within the strategy's type depth, its stop conditions and `Max nodes total`. For a generic interface (`IHandler<T>`) the implementers it can reach are the implementers of every closing (`IHandler<Saved>` , `IHandler<Deleted>`), whether a type implements it directly, through a base class or through another interface.



The Expansion Strategies page: mode, depths, inclusion flags and stop conditions

6.7 Compression rules and pipeline profiles

Two further profiles can be pinned to a template.

Compression rules hold the scoring multipliers and the auto-compression heuristic (accessibility multipliers, priority/role/entry-point boosts, path penalties, dead-code and single-caller weights, the caller-count threshold for full detail, trimming mode, tiebreaker order and the user-hint boost). Three ship with the product:

Profile	Description
`Default (Pareto)`	Mid-range trimming. Recommended for typical code reviews.
`Strict (Aggressive)`	Aggressive token reduction; private and helper members get a strong penalty, tests and generated code even stronger. Suited to solutions that blow the budget.
`Lenient (Conservative)`	Conservative trimming; private and helper members stay relatively heavily weighted. Suited to solutions with a lot of custom logic in private helpers.

Pipeline profiles control the token-budget pipeline and snapshot auto-reload:

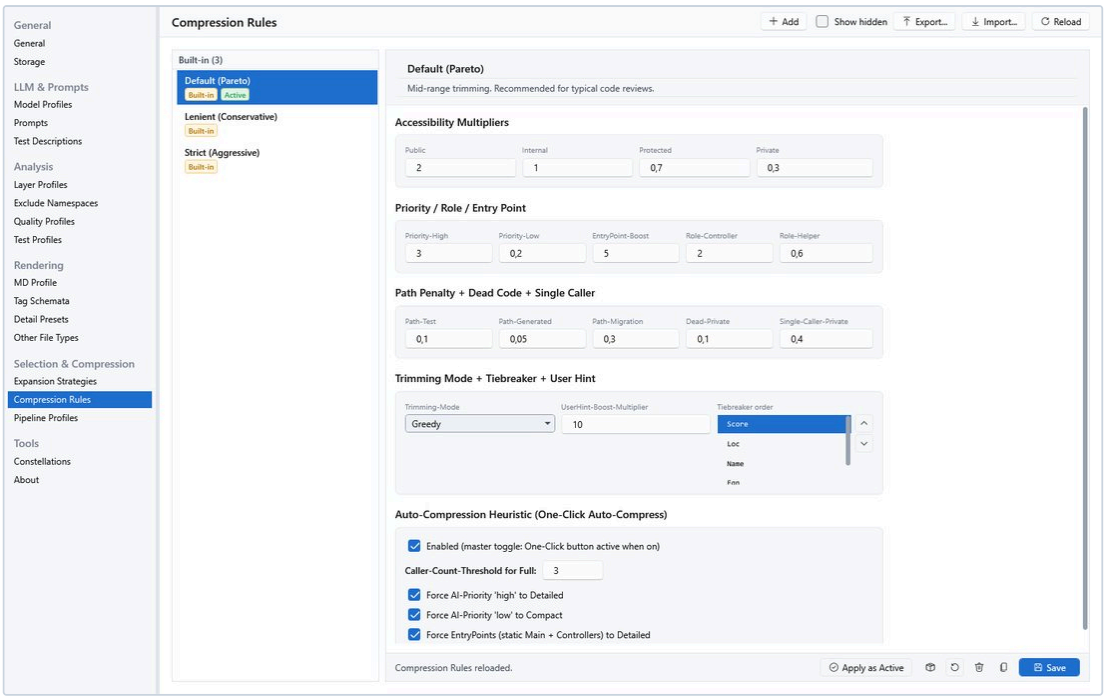
Profile	Budgeted trimming	Max token budget	Max overshoot	Auto-bypass snapshots	Budget header
`Default`	off	60,000	50%	on	off
`Aggressive Trimming`	on	20,000	10%	on	on

Profile	Budgeted trimming	Max token budget	Max overshoot	Auto-bypass snapshots	Budget header
`No Trimming (Snapshot-Mode)`	off	200,000	900%	off	off

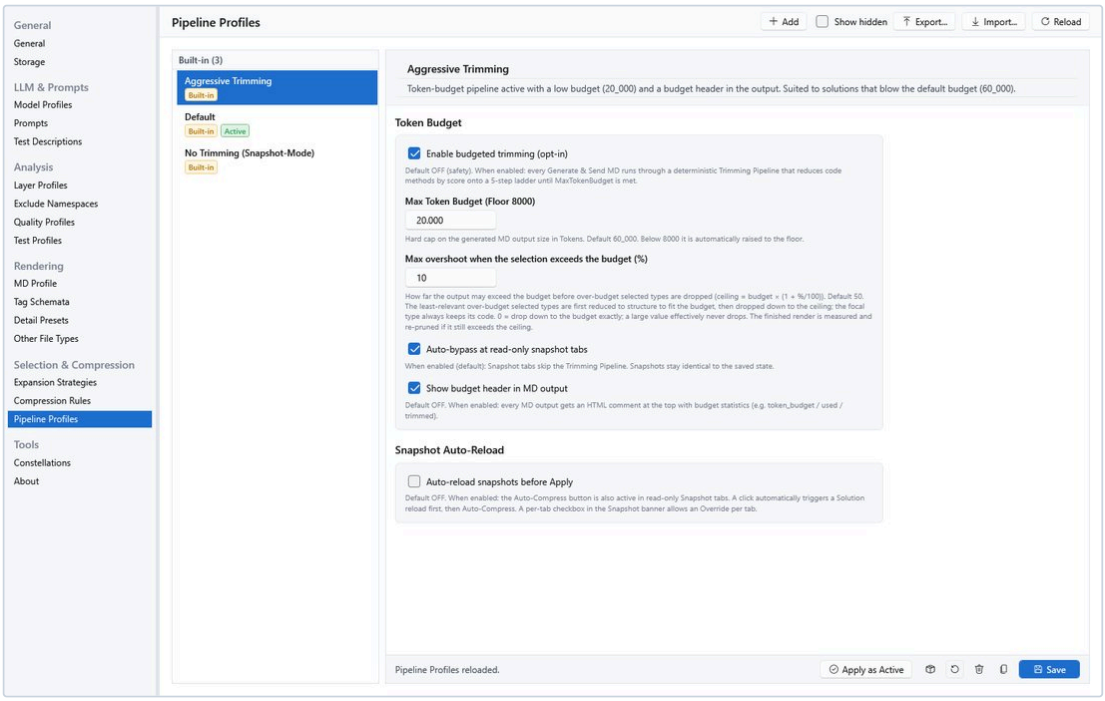
The profile fields in the editor:

Field	Default	Meaning
`Budgeted Trimming`	off	Master switch. When on, every `Generate & Send MD` runs through the trimming pipeline.
`Max Token Budget (Floor 8000)`	60,000	Hard cap on the output size in tokens. Values below 8,000 are raised to the floor.
`Max overshoot when the selection exceeds the budget (%)`	50	How far the output may exceed the budget before over-budget selected types are dropped (ceiling = budget × (1 + percent / 100)). 0 = a hard cap; a large value effectively never drops a selected type. Clamped to 0-1000. The finished render is measured and pruned again if it still exceeds the ceiling; the most relevant type always keeps its code.
`Auto-bypass snapshots`	on	Snapshot tabs skip the pipeline so they stay identical to the saved state.
`Show budget header`	off	Prefixes the output with an HTML comment carrying budget statistics.
`Snapshot Auto-Reload`	off	When on, the auto-compress button is also active in read-only snapshot tabs; a click triggers a reload first.

Note: the `Default` pipeline profile has trimming switched off, so the token-budget machinery runs only when you turn it on - either by activating a profile with trimming enabled, or by setting a token budget on the template or the MCP profile.



The Compression Rules page: multipliers, penalties and the auto-compression heuristic



The Pipeline Profiles page: token budget, overshoot and snapshot behavior

6.8 How a document is assembled

The renderer emits the sections in a fixed order. The `Sections:` block in the document head (`AI_CONTEXT_SPEC`) lists the sections **that document** contains - the full set is larger and depends on the template and the profile.

Section	Controlled by
`SPEC`	slot `Spec`

Section	Controlled by
`META`	slot `Meta`
`AI_PROMPT`	slot `AiPrompt`
`QUALITY_HOTSPOTS`	template switch `Include quality metrics`
`PATH_LEGEND`	non-empty `PathLegend` slot plus the layout legend toggle and a non-empty alias map
`COMPRESSION_LEGEND`	always on
`DOMAIN`	slot `Domain`
`ARCHITECTURE_FLOW`, `INTERFACE_RELATIONS`, `ENTRY_POINTS`, `ENTRY_POINT_FLOW`	graph selection
`SERVICE_DEPENDENCY_GRAPH`, `LAYER_MAP`, `CLASS_DEPENDENCY_GRAPH`, `ROLE_GRAPH`, `METHOD_GRAPH`, `METHOD_USED_BY_GRAPH`	their graph slots
`ENUM_SUMMARY`	slot `EnumSummary`
`FILE_INDEX`	slot `FileIndex`
`BUILD_AND_UI_FILES`	always on, relevance-filtered
`AUXILIARY_FILES`	always on, relevance-filtered
`SOLUTION`, `PROJECT`, `FILE`	fixed structure
`CLASS`, `INTERFACE`, `ENUM`	staged slots
`QUALITY_FINDINGS`	rendered when findings exist

Two facts about the render step are worth knowing:

- **Detail reduction happens at render time, not during analysis.** The analysis always produces the full object graph; the renderer decides what reaches the Markdown. Changing a detail setting therefore only needs a new render, not a new analysis.
- **The document describes itself.** Besides the section list, the head carries the alias rules, the graph edge semantics (`A -> B` means "A depends on B", `A implements B`, `A <- B` means "A is used by B"), and the dependency confidence levels (`[high]`, `[medium]`, `[low]` are relative heuristics for the model, not formal proof levels).

Type aliases and the path legend

When the layout legend is on, long type names are abbreviated and a `PATH_LEGEND` table decodes them:

- A type name longer than 12 characters is abbreviated to the initial letters of its capitalized words. At least two characters are used; a name with fewer than two capitals falls back to its first three characters, uppercased.
- A collision between two abbreviations gets a numeric suffix, so `CountingAnalyzer` may become `CA` and `CodeAnalyzer` `CA2`.
- Names of 12 characters or fewer are left unchanged.
- The map is keyed by the simple type name, so two types that share a simple name share one alias.

The aliases are substituted in ten sections: `DOMAIN`, `ARCHITECTURE_FLOW`, `INTERFACE_RELATIONS`, `ENTRY_POINTS`, `ENTRY_POINT_FLOW`, `SERVICE_DEPENDENCY_GRAPH`, `LAYER_MAP`, `CLASS_DEPENDENCY_GRAPH`, `METHOD_GRAPH` and `METHOD_USED_BY_GRAPH`. Source-code blocks are never rewritten - their contents are literal C#.

The compression legend

Every document carries a `COMPRESSION_LEGEND` block that explains how methods are written:

- Full mode: all sub-blocks (`METHOD_INFO`, `DEPENDENCIES`, `USED_BY`, `SEMANTICS`, `SUMMARY`, `AI_TAGS`, `METHOD_CODE`).
- Compact mode (`compact="isolated"`): a single line with signature, return type, access and role. A missing `compact=` attribute means full mode.
- The selection rule: primary-flow methods render full, isolated-flow private helpers render compact; per-node overrides and auto-compression heuristics can shift any method.
- The provenance vocabulary of a `SEMANTICS` block: `Resolved` marks values from the resolved analysis (alone when no per-field provenance was recorded, or broken down as `Source: Resolved (field=token, ...)`), with the tokens `fact`, `ai`, `inferred` and `verified-absent`.
- A caveat on caller lists: they name only the callers this document renders, and they are statically resolved - reflection, DI containers, serializers and external assemblies stay invisible. Absence of callers is not proof of dead code.

Line numbers, quality metrics and file content

With `Include line numbers` on, every method and type tag gets a `lines="START-END"` attribute (1-based, the same convention as the IDE status bar). With `Include quality metrics` on, the document gains the `QUALITY_HOTSPOTS` section (the most complex methods and most coupled types) and each method/type tag is annotated with `complexity=""` and `ce=""` (efferent coupling).

`BUILD_AND_UI_FILES` renders each included project's `.csproj` plus the XAML files tied to the included code slice. XAML is emitted in a compressed structural form; the detail level decides the form: at Compact only the root tag with a child/attribute summary, at Normal the root plus the direct child tag names, at Detailed the whole tree (comments removed), and at Source the file content as it stands. `AUXILIARY_FILES` renders project-wide structure files always and loose config/documentation files when the prompt names them.

6.9 Markup analysis: XAML and AXAML

WPF and Avalonia bindings and resource keys are strings resolved at runtime. The compiler does not check them, and the C# symbol graph cannot see them, so a member consumed only from markup looks unused. AICB scans the markup textually and records the edges it can prove.

What is scanned

- Both dialects: WPF `.xaml` and Avalonia `.axaml`. The extraction patterns differ per dialect, because Avalonia accepts markup WPF does not (for example `{CompiledBinding}`).
- Discovery uses the same file walk as the analysis, with `bin`, `obj` and `.vs` excluded. A folder of that name that contains the project itself - a checkout under `~/bin/repo`, for example - is the checkout's location, not build output, and does not exclude anything. Each file is parsed once and shared by all extraction passes.
- Every match is mapped to a 1-based line, and markup inside an XML comment is ignored - a commented-out binding must never produce a phantom fact or a false duplicate key.

- A view is labeled as a solution-relative path with a dialect suffix, for example `UI/Views/FooView.xaml (xaml)` or `MainWindow.axaml (axaml)`.

Binding scopes: what is checked and what is skipped

A binding is only checked when the DataContext it resolves against can be typed with certainty. Four scopes exist:

Scope	Meaning	Checked?
Page view-model	The file-level DataContext, resolved from `d:DesignInstance` or from the `FooView` → `FooViewModel` naming convention.	yes
Named type	A `DataTemplate` with a resolvable `DataType`; the binding targets that item type.	yes
Unknown	A context that cannot be typed from the text: a nested runtime DataContext, a `DataType`-less template, a `ControlTemplate` or a `Style`.	no - skipped, never reported
Collection item	A binding on the row object of an items host, recovered from the host's `ItemsSource={Binding Member}`. Two shapes: a DataGrid/GridView column value binding, and everything inside a `DataType`-less `*.ItemTemplate`.	yes, once the collection's element type resolves; otherwise treated like Unknown

This is deliberate: a scope that cannot be typed is skipped silently rather than guessed. The result is a possible false negative, never a false report.

Unresolved bindings

`find_unresolved_bindings` lists the silently broken bindings: a `{Binding X}` whose root member does not exist on the confidently resolved DataContext view-model - a typo, a removed or renamed member, or a wrong DataContext. The compiler does not catch this class of defect. Each report carries the view, the resolved view-model, the missing member, and the 1-based line of the attribute carrying the binding.

The same findings surface as the insight `design-unresolved-xaml-binding` (severity Warning, producer group Design Smells). The producer can be switched off in the active Insights profile.

Notes:

- The facts are computed during analysis and are live-only. A recalled database snapshot reports none; re-run the analysis first.
- Only view-models resolved with certainty and with a verifiable member surface (an in-solution base chain or a known MVVM base) are checked. Source-generated members (`[ObservableProperty]`, `[RelayCommand]`) and inherited members count as resolved.
- Read a hit as high-confidence evidence to verify at its file and line, not as proof: the residual risk is scope typing, not member lookup.
- A report is not a completeness claim about your markup. The answer carries the population it looked at - `markupFilesScanned` and `bindingSitesInMarkup` - so "no broken bindings among 1,021 sites in 164 markup files" and "no markup at all" are distinguishable.

Avalonia: explicit data types only

On an `.axaml` view, the DataContext type is resolved **only** from `x:DataType` at scope level - never through the WPF naming convention, which would guess wrong under Avalonia's naming. A page without `x:DataType` is simply untyped: its bindings are skipped, never flagged. An `x:DataType`-less template, including a `TreeDataTemplate`, is always skipped.

Markup type references and the member fan-in

Markup references types and members, and those references are folded into the symbol graph, so they show up in `find_usages` and `impact_of_change`:

- **Type references** are recorded in four forms: the `{x:Type ns:T}` form (a `Style` or `ControlTemplate` `TargetType`, a `DataType`, an `x:Type` setter value), the element-syntax form (`<controls:Arc/>` instantiation or a property-element owner), the attached-property owner form (`mah:SliderHelper.EnableMouseWheel` in a setter, an attribute or an `{x:Static}` owner) and the custom markup-extension form (`{prefix:Name}` → the type `NameExtension` or `Name`). This keeps a type that is referenced only from markup - a control whose default style lives in a theme dictionary, an attached-property helper, a markup extension - out of the dead-code report. A nested type written `prefix:Outer+Inner` is read as that nested type in these forms and in `x:Static`, `DataType` / `x:DataType`, `TargetType` and `d:DesignInstance`, so bindings under such a scope are judged against the nested type.
- **Member references** are recorded for event wires, `{x:Static}` members, attached-property read accessors and `{TemplateBinding}` members (resolved against the enclosing template or style target; an unresolvable target, a parenthesized attached path and a prefixed path with an unresolved prefix are declined).
- **Event wires** (WPF `.xaml` and Avalonia `.axaml` alike; `Handler` in `Click="Handler"`, `Loaded="OnLoaded"`) count for every event the element's type or one of its bases declares, for a delegate-typed property such as `EventSetter.Handler`, and in the `Owner.Event` form for an attached event the owner declares. Where the element's type cannot be resolved, only `Click` is read.
- **Property sites** (`.xaml` and `.axaml` alike) record an edge on the property they name: an attribute or a property element that sets a property, an `Owner.Member` attached property, the `Property=` of a `Setter` (a set) and of a `Trigger` or `Condition` (a read), and an attached path under a non-`clr` prefix in `{Binding}` or `{TemplateBinding}`. A property without an owner is read against the enclosing `Style` or `ControlTemplate` `TargetType`. The edge lands on the property's CLR wrapper, and for an attached property on `SetX` where markup sets it and on `GetX` where it reads it; the declaring type must be in the solution. A `TargetName` setter and a `SourceName` trigger name a property of a named part and are declined. Under an Avalonia `<Style Selector="...">`, a bare `Setter Property=` names a property of the type the selector's last segment names (`Button` in `Button.accent` or `Button:pointerover`, `ContentPresenter` in `^:pointerover /template/ ContentPresenter#PART`, a prefixed `ns|Type` through its prefix), and a nested `^` style inherits its parent's target. A selector that names several types or none (a comma, only a class, a name or `:is()`) records nothing, and an Avalonia `Classes.accent="True"` toggle is not counted as a use of a `Classes` member.
- Bindings are recorded as `{Binding}` root paths, attributed to the resolved view-model and tagged `(xaml)` as low confidence. A view-model reached only through a design hint or the naming convention records its member edges but not its type edge. For this fan-in, a WPF view that has 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 that is assigned two different types gets no view-model. The unresolved-binding check does not use this source.
- **Safe direction:** a non-solution prefix and an unprefixed framework element produce no edge. The scan would rather miss a reference than invent one.

Resource keys

`find_resource_usages` answers the resource-key question. With a key it reports where the key is defined (`x:Key` , with file and line), every reference site, and the reference kind:

Kind	Meaning
`static`	`{StaticResource Key}` - resolved once, at load time.
`dynamic`	`{DynamicResource Key}` - re-resolved on every lookup, so it survives a theme swap. A theme-swapped brush must be dynamic.
`code`	A C# string literal whose text equals the key - the `TryFindResource("Key")` channel. Reported with file and line, because a matching literal is not by itself proof of a resource lookup.
`theme`	WinUI's `{ThemeResource Key}` , re-resolved when the theme changes.
`callerMemberName`	A C# member whose name is the key, read through a `[CallerMemberName]` accessor that indexes a resource dictionary or calls `FindResource` with that parameter. The site is the call.

A markup extension the solution itself derives from `StaticResourceExtension` or `DynamicResourceExtension` (for example `{ui:ThemeResource Key}`) counts as the kind of its base, `static` or `dynamic` ; one that ships in a referenced library is not recognized. The answer for a key adds `themeReferenceCount` and `callerMemberNameReferenceCount` when they are not zero.

Without a key you get the audit view: every never-referenced key (a deletion candidate) plus same-file duplicate definitions. A same-named key in a different file is usually a legitimate theme pair and is not reported as a collision.

Notes:

- String keys only. A structural `x:Key="{x:Type ...}"` and the `{StaticResource ResourceKey=...}` long form are out of scope.
- A key assembled at runtime (interpolation, concatenation, a constant referenced by name) is invisible.
- In C# sources, comments are not excluded: over-reporting a quoted key inside a comment costs one visible line, while wrongly hiding a real literal could turn a live key into a deletion candidate.

The markup tools

Tool	Answers	Key arguments	Limits
`find_binding_usages`	Which markup sites bind a member path, and what each one resolves against.	`path` (omit for the inventory of all bound root members), `scope` (view-path substring), `form` (`dataContext` , `elementName` , `relativeSource` , `source`).	The `{Binding}` markup-extension form only - element syntax `<Binding Path="..."/>` , WinUI `{x:Bind}` , `{TemplateBinding}` and bindings built in code-behind are invisible. Lists are capped at 200.

Tool	Answers	Key arguments	Limits
`find_resource_usages`	Key definitions, reference sites and kinds; the never-referenced audit; same-file collisions.	`key` (omit for the audit view), `scope` (view-path substring).	String keys only; markup, C# literals and `[CallerMemberName]` accessors; capped at 200.
`find_unresolved_bindings`	Broken `{Binding}` paths against a confidently resolved view-model.	none beyond the session.	Live analysis only; conservative skips; see above.

Two behaviors of `find_binding_usages` are worth knowing:

- A path matches as the whole path **or** as its root segment. Asking for `Foo` finds `{Binding Foo}`, `{Binding Foo.Bar}` and `{Binding DataContext.Foo, RelativeSource=...}` - all three consume the member `Foo`. The `Bar` in `{Binding Foo.Bar}` does not match; those sites are reported separately as deeper-segment sites, because that `Bar` is a member of `Foo`'s type. Matching is case-sensitive.
- 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}}` is a site of `Cmd`. A hop in the middle of a path (`PlacementTarget.DataContext.Cmd` in a context menu) is not followed.

Both scanning tools read the current files on disk on every call, so their answers are fresh after a markup edit and no refresh is needed; they also work on recalled sessions. The bindings they cannot see are listed in their own answers, and `find_binding_usages` reports the number of pathless bindings (an empty `{Binding}`, a parameters-only `{Binding Mode=OneWay}`, or an attached-property path) so the listed sites and the markup's binding count differ visibly rather than silently.

7 Profiles, master data and solution configuration

AICB is steered by named, editable configuration entries. They fall into two groups: **profiles**, which decide how something is done (which tools an MCP client sees, which quality checks run, how a document is rendered), and the remaining **master data**, the building blocks a profile or a template is assembled from. This chapter describes the entry model, the kinds that exist, how a configuration is pinned per solution, how the `.aicb.json` sidecar carries it into your repository, and in which order AICB resolves a value when several places define one.

7.1 Master data and profiles

What every entry shares

All library entries share a stable identifier, a `Name`, and the list markers described below, and they use the same management gestures. Many entry types also have an optional `Description`; it is not a universal field (for example, tag schemata and test descriptions have their own type-specific content instead).

Built-in entries, your own entries, and the pills

- A **built-in** entry ships with the application. It cannot be removed permanently.
- A **custom** entry is one you created, either with `Add` or with `Duplicate`.
- Editing a built-in and pressing `Save` does not destroy it: the entry is marked as overridden and shows the `Overridden` pill. AICB then leaves the entry alone when a later version refreshes the built-in defaults. The hint above the editor states it plainly: `Built-in profile - edits will be marked overridden. Use Restore to revert to code defaults.`
- `Restore Built-In` resets a built-in to its code defaults and clears the override/hidden flag. The tooltip reads: `Restore Built-In: reset this built-in to its code defaults and clear the override/hidden flag.` The action is available whenever the selected built-in is overridden or hidden.
- `Delete` on a built-in **hides** it instead of removing it (the `Hidden` pill appears in the list once you tick `Show hidden`). Hidden entries are invisible in the normal list; with `Show hidden` switched on they reappear and can be brought back with `Restore Built-In`. `Delete` on a custom entry removes it permanently.
- `Duplicate` creates a custom copy of the selected entry with a new identity and the name `<name> (Copy)`. This is the usual way to adapt a built-in without touching it.
- `Save` persists the editor values; `Reload` re-reads the library from the database and discards unsaved editor changes. The pages offer the same keyboard shortcuts throughout: `Ctrl+S` for Save, `Ctrl+D` for Duplicate, `Ctrl+N` for Add, `F5` for Reload.
- Each library page can `Export...` all of its entries to a JSON file and `Import...` them back. Entries marked as built-in in the import file are always skipped. For custom entries, conflicting ids can be skipped or replaced; in `Replace` mode a custom entry whose id collides with a built-in id can overwrite that built-in row, so use `Skip existing` for untrusted files.

The list rows carry up to six pills:

Pill	Meaning
`Built-in`	The entry ships with the application.
`Overridden`	You edited a built-in; `Restore Built-In` returns it to its code defaults.
`Hidden`	A built-in you soft-deleted; visible with `Show hidden` and recoverable with `Restore Built-In`.

Pill	Meaning
`ACTIVE`	This is the globally active entry of a family that has exactly one active member. `Apply as Active` moves the marker.
`DEFAULT`	This is the default entry where a family may have several defaults (detail presets, model profiles, expansion strategies, tag schemata per section type).
`IN USE · N`	Other master entries reference this custom entry `N` times. The pill's tooltip says references must be removed before deleting; treat it as a warning to check before you delete, because nothing in the library actually blocks the deletion.

Where the libraries are edited

Most libraries live on the `Settings` page, in the left sub-navigation. The 18 entries are visually grouped under the non-clickable headers `General`, `LLM & Prompts`, `Analysis`, `Rendering`, `Selection & Compression` and `Tools`. Three further master-data pages live outside `Settings`: `Templates` (under `Configuration`), `Run Templates` (under `Workspace`) and `MCP Profiles` (under `MCP`).

The kinds of profiles and master data

Page	What it controls
`Model Profiles`	A named LLM configuration: endpoint, model name, token limits and optional prices. `Test Connection` checks the profile against the real endpoint. Built-in profiles ship **without** prices, so cost estimates appear only after you fill in the optional pricing fields yourself.
`Prompts`	Reusable user-prompt presets that context templates reference.
`Test Descriptions`	Reusable acceptance-test descriptions for run templates. Built-in descriptions: `General Code Review`, `SOLID Violations`, `Layer Violations`, `Pipeline Output Validation`, `Security Smells`, `Performance Hot Path`, `Async Pattern Compliance`, `MVVM Bindings Audit` and `SAP DI API Patterns`. The SAP one audits SAP Business One DI API and Service Layer usage: missing COM-object releases (`Marshal.ReleaseComObject`), leaked form handles, missing error checks after DI API calls, and DI calls without retry or error propagation. Each finding names the `SAPbobsCOM` / `SAPbouiCOM` method, the risk and the recommended pattern (a `using` block, or `try` / `finally` with `ReleaseComObject`).
`Layer Profiles`	Namespace-pattern → architecture-layer mappings. Rules are evaluated in list order, the first match wins; an empty rule list means no namespace matching at all and the role-based heuristic applies. Each profile also carries a `Layering Policy`: `Advisory` reports layer violations as warnings, `Strict` as critical findings. Built-in profiles: `No mapping`, `Clean Architecture`, `Onion Architecture`, `Domain-Driven Design`, `Hexagonal Architecture`, `MVVM`, `Vertical Slice Architecture` and `SAP Business One Add-On`. The SAP profile maps `.Forms.` and `SAPbouiCOM` to Presentation; `.AddOn.`, `.AddOns.`, `.EventHandlers.`, `.MenuHandlers.` and `.BusinessLogic.` to Application; and `SAPbobsCOM`, `.DataLayer.`, `.ServiceLayer.`, `.SLClient.` and `.DIApi.` to Infrastructure.
`Exclude Namespaces`	Named lists of namespace patterns the analyzer skips. Built-in lists: `None (do not exclude any namespace)`, `BCL default (System. / Microsoft. / Windows. / Syncfusion.)` and `SAP Business One default (SAPbouiCOM. / SAPbobsCOM.)`. The BCL preset also excludes `CommunityToolkit.*`; that rule is not repeated in its shorter display name.

Page	What it controls
`Quality Profiles`	Switches the individual quality checks (the insight producers) on and off and holds their thresholds (method length, cyclomatic complexity, member counts), dismissal policy, action mode and optional send-template id. Debt costs are fixed by the individual producers, and the CLI gate threshold comes from `--fail-on`; neither is configured by the profile. A quality profile decides **whether** a check runs; the severity of its finding is a separate axis.
`Test Profiles`	Which projects count as test projects and which method attributes mark a test case. Built-in presets: `Default`, `xUnit`, `NUnit`, `MSTest`. The `Default` preset replicates the built-in heuristic exactly: a project name containing `Test`, `Tests`, `Spec` or `Specs` is a test project, and `Fact`, `Theory`, `Test`, `TestMethod`, `TestCase` mark a test case (matched as substrings, so `WindowsFact` matches through `Fact`).
`MD Profile`	A container with one slot per output-section type. There are 27 section types (6 of them graph sections); four are staged slots (`Class`, `Method`, `Interface`, `Enum`). In each slot you choose a tag schema. The editor has the tabs `Overview`, `Non-graph slots`, `Graph slots` and `Referenced by`. Note: the label is singular even though the page manages several profiles.
`Tag Schemata`	The actual rendering configuration of one output-section type: which tags, in which order, in which layout. Per section type there are several built-in schemata and optionally your own; exactly one is the default for its type. The MD profile is the mapping, the tag schema is the content.
`Detail Presets`	Describes **how** content is compressed at one detail level. The detail level (`Compact`, `Normal`, `Detailed`, `Source`) says how much; the preset says how. A context template has four slots, one per level.
`Other File Types`	A read-only reference page: the recognized file types and the four detail stages. The compression stage itself is chosen per node in the Solution Tree.
`Expansion Strategies`	How far the selection reaches from a marked tree node: method depth, type depth and flags. The expansion strategy decides **what goes into** the context; a compression rule decides how short it is then written.
`Compression Rules`	A named rule set for how strongly code blocks and sections are shortened when the token budget gets tight.
`Pipeline Profiles`	The token-budget trimming configuration.
`Constellations`	Import and export bundles of master data plus portable settings as a single JSON file.
`General`, `Storage`, `About`	Plain settings and information pages, not libraries. `General` holds the app-wide preferences and the global defaults; `Storage` holds the database location and backup settings.
`Templates`	**Context templates**: the configuration root of an export. A context template binds a prompt, an MD profile, the four detail-preset slots, the four expansion-strategy slots, an optional test description and the profile overrides. The editor sections are `Prompt & MD profile`, `Selection engine (per detail level)`, `Export content` and `Profile overrides`.
`Run Templates`	**Run templates**: the configuration of a run. A run template selects a context template and adds the per-run-type configuration. Three run types are available in the editor (`Manual`, `Iteration`, `Preselection`); the `Pipeline` run type is planned and not yet released, so it is not offered.
`MCP Profiles`	Which tools an MCP client sees, plus the render notation, the token budget and the automatic-refresh behavior. Exactly one MCP profile is active; it drives the MCP server.

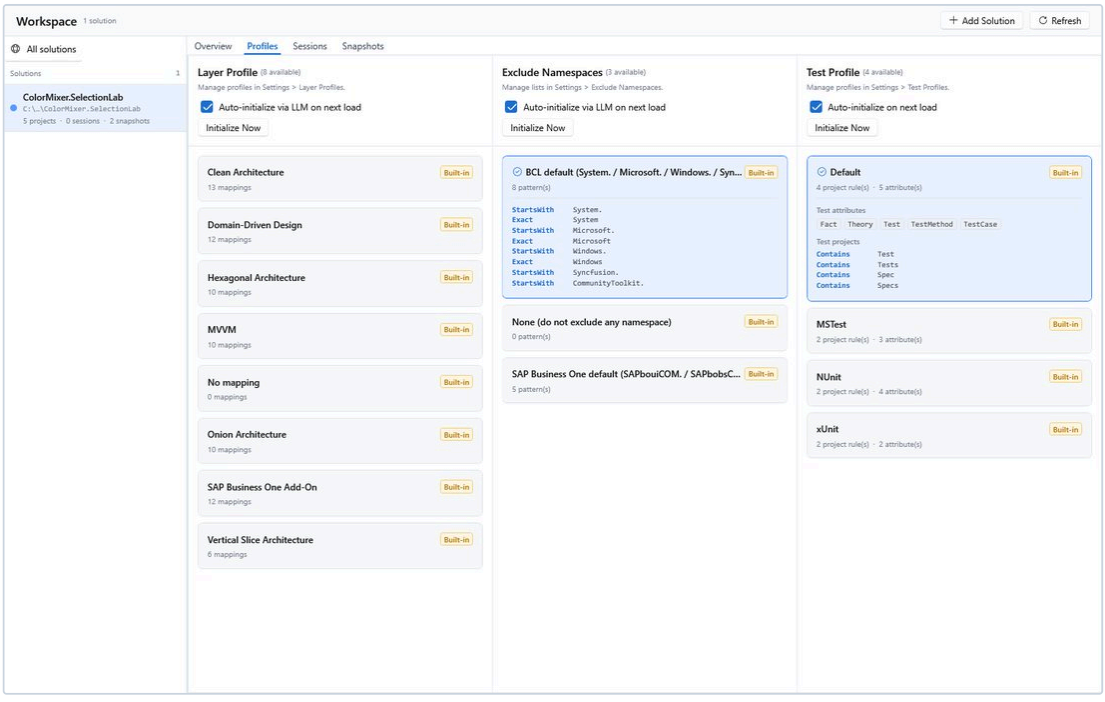
Note: `Templates` and `Run Templates` are two different things with similar names. A context template is the configuration root of an export; a run template is the configuration of a run and merely **selects** a context template. On screen the labels are `Templates` and `Run Templates`.

7.2 Per-solution configuration

The three axes in `Workspace`

Select a solution on the `Workspace` page and open the `Profiles` detail region. It shows three pickers side by side, one per configuration axis: `Layer Profile`, `Exclude Namespaces` and `Test Profile`. Each picker chooses the entry that is active for the selected solution.

- The header shows how many entries the library offers, for example `(12 available)`.
- A card marked `Built-in` is a shipped preset.
- The selected card unfolds a rule preview: a layer profile shows its rules as `pattern > Layer`, an exclusion list as `MatchType pattern`, a test profile its project rules.
- The pickers only choose. Creating and editing happens in the `Settings` libraries, and each picker points there, for example `Manage profiles in Settings > Layer Profiles.`
- Without a selected solution the pickers are inert and show the global default; the exclusion picker explains why: `Select a solution to choose its active list.`



The per-solution profile pickers on the Workspace page

The precedence for each axis is:

Axis	Resolution
Layer profile	per-solution choice > global default (`Settings > General > Application`, `Default layer-mapping profile`) > role-based heuristic
Exclusions	per-solution choice > global default (`Default namespace-exclusion list`) > built-in `BCL default`

Axis	Resolution
Test profile	per-solution choice > globally active test profile (`Settings > Test Profiles` , `Apply as Active`) > built-in `Default`

An exclusion list can hold **keep rules** besides its exclusion rules (the `Keep` column of the list editor): a namespace a keep rule matches is never excluded. A list made only of keep rules does not replace the list beneath it - a solution's keep-only list applies over the global default, and a global keep-only list over the built-in `BCL default` - so a library whose subject is a framework area (Dapper and `System.Data`) needs one rule. A list with exclusion rules of its own replaces the list beneath it, as before. Whatever list applies, a type the solution declares itself is never excluded because its namespace matches, and the document header names the list it was analyzed with (`Exclusions:`).

First-time setup on load

Each picker carries a checkbox: `Auto-initialize via LLM on next load` for the layer profile and the exclusion list, `Auto-initialize on next load` for the test profile. A freshly registered solution starts with all three flags enabled. They are evaluated the next time you open that solution in the Context Builder - the loading happens there, not in the `Workspace` tab.

When such a solution is loaded and an armed axis is still unconfigured, AICB offers to set it up:

- If the sidecar covers the axis, it is **restored**, with no LLM call. The created entry carries the description `Restored from the .aicb.json sidecar` ; its name stays `<solution> - layers` , `<solution> - exclusions` or `<solution> - tests` .
- Otherwise the axis is **generated**: with a usable default model profile, the layer rules and exclusions come from one LLM call over the solution's declared and referenced namespace lists; the test axis comes from a local heuristic over the analysis. The created entry carries `Auto-initialized on load` . If no model profile is available or no tests are detected, the corresponding proposal may remain unavailable.
- A restore asks for confirmation before it is applied. For generated layer/exclusion proposals, the LLM request is made first and the resulting proposal is then shown for confirmation; declining prevents the database change but cannot undo the request. An axis that already has a choice is never overwritten, and after a successful initialization the checkbox is cleared.

Successful first-load setup creates and activates per-solution entries in the local configuration database. It does **not** write the sidecar. Use `Workspace > Profiles > Export Config` afterwards if the configuration should travel with the repository.

`Initialize Now` in each picker applies a stored configuration immediately, without waiting for a load and without any analysis: it restores the axis from the sidecar. If there is nothing to restore - the axis is already configured, or the sidecar does not cover it - a dialog says so.

Guided setup through an agent

Three MCP tools drive the same configuration:

- `solution_config_status` reports whether each of the three axes is initialized, which entry is active and where the active value comes from (`db` , `sidecar` or `none`). "Initialized" is an explicit marker set by `apply_solution_config` ; a repository with a valid sidecar but no database row reads as initialized with source `sidecar` .

- `init_solution_config` returns the proposal material: the solution's own (declared) namespaces for layer rules, its referenced namespaces for exclusion rules, and the detected test projects plus the test-attribute markers that actually occur in the code.
- `apply_solution_config` applies a proposal: it creates custom entries, activates them, stamps the axes as initialized, and writes both the configuration database and the `<SolutionName>.aicb.json` sidecar next to the solution. It returns the sidecar path. An invalid `matchType` is rejected rather than silently corrected, and the test axis can alternatively be cloned from a built-in preset (`xunit` , `nunit` , `mstest` , `default`).

`check_solution_config_drift` checks whether a configuration still fits the code: declared namespaces that no layer rule maps, referenced namespaces that no exclusion covers, and test projects the active test profile does not match. Only a custom configuration is evaluated; a built-in preset counts as a deliberate generic choice, not as drift.

Marking findings as intentional

A finding you decide is by design becomes a **suppression**. In the Insights panel, the eye-off button on a line records it. Its tooltip describes the effect: `Mark this finding as intentional - it stops being reported here and to the MCP. The quality gate (--fail-on / solution_metrics) still counts it. Export the solution config to share the decision via the .aicb.json sidecar.` A suppression can address a whole check, one type, or one member, and it can carry a reason. Suppressions are per solution and reach your team through the sidecar. The reset button in the Insights toolbar (`Reset what you hid`) brings dismissed findings and your own suppressions back into view, while suppressions declared in the committed sidecar stay in force.

7.3 The `.aicb.json` sidecar

What it is and where it lives

The sidecar is a configuration file next to your solution file, named after it: `<SolutionName>.aicb.json` . For `MyApp.sln` the file is `MyApp.aicb.json` in the same folder. It is meant to be committed to your repository, so the solution configuration is versioned with the code. Headless hosts consume the supported fallback axes directly; the desktop app can restore its profile content into the local database.

The file carries the **content** of a configuration - patterns, names, rules - and no database identifiers. That is what makes it portable: it is not tied to the database of one machine.

The complete schema

```
{
  "layerRules": [
    { "pattern": ".Application.", "matchType": "Contains", "layer": "Application" }
  ],
  "layeringPolicy": "Strict",
  "exclusions": [
    { "pattern": "System.", "matchType": "StartsWith" }
  ],
  "keepNamespaces": [
    { "pattern": "System.Data", "matchType": "StartsWith" }
  ],
  "testProjectRules": [
    { "pattern": ".Tests", "matchType": "EndsWith" }
  ],
  "testAttributeNames": [ "Fact", "Theory" ],
  "autoInit": { "layer": false, "exclusions": false, "test": true },
  "suppressions": [
    { "producer": "quality-long-methods", "type": "LegacyImporter", "member": "Import", "reason":
"generated" }
  ],
  "analyzePreferredTfmOnly": true,
  "designTimeBuildPlatform": "x64"
}
```

Key	Form	Meaning	When omitted
`layerRules`	array of `{ pattern, matchType, layer }`	Namespace pattern → architecture layer. The first matching rule wins.	The layer axis is unset. `matchType` defaults to `Contains`.
`layeringPolicy`	`"Advisory"` or `"Strict"`	How a layer violation is reported. `Advisory` = warning, `Strict` = critical (acts as a gate).	`Advisory`. The key is only written when the value is not `Advisory`, so an advisory file stays byte-identical to older files.
`exclusions`	array of `{ pattern, matchType }`	Namespace patterns the analysis skips.	The exclusion axis is unset, and so it is for an empty array `[]`: the list beneath applies. `matchType` defaults to `StartsWith`.

Key	Form	Meaning	When omitted
`keepNamespaces`	array of `{ pattern, matchType }`	Keep rules: a namespace they match is never excluded. Alone, they keep the built-in list in force beneath them.	No keep rules. Versions before 0.5.501.1 ignore this key and drop it when they rewrite the file.
`exclusionsDisabled`	`true`	The opt-out: exclude nothing. Written for a solution set to exclude nothing, and for a list whose rules are all empty.	The exclusion axis follows `exclusions`.
`testProjectRules`	array of `{ pattern, matchType }`	Rules that classify a **project name** as a test project; any match counts.	The test axis is unset. `matchType` defaults to `Contains`.
`testAttributeNames`	array of strings	Attribute names that mark a method as a test case, matched as substrings.	The test axis is unset.
`autoInit`	`{ layer, exclusions, test }` booleans	The "auto-initialize on next load" opt-ins per axis, so the intent travels with the repository. The database record remains the primary source; this is the fallback for a fresh clone.	Written only when at least one flag is `true`.
`suppressions`	array of `{ producer, type?, member?, reason? }`	Triage decisions. `producer` alone silences a whole check, `producer` + `type` every hit on one type, `producer` + `type` + `member` exactly one member. `reason` is free text for reviewers.	No suppressions are declared.
`analyzePreferredTfmOnly`	`true` or `false`	Analysis scope: analyze only the preferred target framework's instance of a multi-targeted project.	The scope is not declared. An explicit `false` is a deliberate statement and is written back as `false`.
`designTimeBuildPlatform`	string, a platform as the solution names it (`"x64"`, `"ARM64"`, `"Any CPU"`)	The platform every project is design-time built at, as `dotnet build -p:Platform=x64` would build it. Applies to the desktop app, the CLI and the MCP server. A solution platform is passed as the project platform it maps to under the solution's default configuration. A value the solution file does not name, or a solution platform under which the solution builds its C# projects at several platforms, is refused: the load falls back to the built-in rule, and `get_diagnostics` and the CLI disclose the refused value. If the solution file cannot be read, the value is applied and disclosed as not checked. A blank value, or one that is not a string, makes the whole file invalid, so it is ignored entirely.	The built-in rule decides (see "The build platform"). Versions before 0.5.503.1 ignore this key; 0.5.501.1 and 0.5.502.1 keep it when they rewrite the file, 0.5.500.1 and older drop it.

The allowed `matchType` values are `Contains`, `StartsWith`, `EndsWith` and `Exact`; on reading they are matched case-insensitively. `apply_solution_config` rejects an invalid value; in a hand-edited file an unparsable value falls back to the axis default.

Note: `layeringPolicy` belongs to the layer axis, `keepNamespaces` and `exclusionsDisabled` to the exclusion axis, and `testProjectRules` plus `testAttributeNames` together are the test axis. That makes eleven keys but seven axes: layer, exclusions, test, auto-init, suppressions, analysis scope and build platform. A keep rule written inside `exclusions` (`"keep": true`) is refused as a broken file; `apply_solution_config` moves such a rule to `keepNamespaces` when it rewrites the file.

The analysis scope, written by hand

`analyzePreferredTfmOnly` has no wizard and no control in the graphical interface, so it is written by hand (`designTimeBuildPlatform` is the other such key, see "The build platform"):

```
{ "analyzePreferredTfmOnly": true }
```

A multi-targeted project (`<TargetFrameworks>net8.0;netstandard2.0</TargetFrameworks>`) is loaded once per target framework, so every source file is analyzed N times. With this key set, AICB analyzes only the newest target framework's instance of each project, and the export contains one copy of each project instead of N.

The symbol inventory is unchanged - every symbol a query can name is still there. Two things do change: a fan-in edge whose only source is a non-preferred instance is lost, so `find_usages`, `impact_of_change` and `call_graph` can report a smaller blast radius; and transitive side-effect facts shift. The default is off, and on a solution without multi-targeting the setting does nothing at all.

Creating and updating the file

- In the graphical interface, select the solution in `Workspace` and use `Export Config`. It writes the active layer profile, exclusion list, test profile, your triage decisions and the auto-init flags to the sidecar. If the file already exists you are asked before it is overwritten; then it keeps the suppressions only the file holds (but not one you lifted in the Insights panel) and the file's settings for every axis the solution has no choice of its own for, and the confirmation names what it kept and counts what it wrote. A file it cannot read is left untouched and the export is refused. The confirmation reminds you to commit it. Headless analysis auto-discovers the sidecar without `--db-path`, but applies each field according to the per-axis matrix below rather than treating the whole file as one profile.
- `Import Config` reads the layer rules, the exclusions with their keep rules and `exclusionsDisabled`, and the test axis from a selected JSON/sidecar file and applies them to the selected solution: it creates and activates a custom layer profile, exclusion list and test profile, and keeps a `Strict` layering policy. It does not import the suppressions, auto-init flags, analysis-scope key or platform key.
- `apply_solution_config` writes the configuration database and the sidecar together and returns the path. It keeps the existing file's settings for every axis the call does not set. When the existing file cannot be read, the call is refused before anything is written; a file that becomes unreadable while the database is written is left alone, and a warning says the file was not written.
- You can edit the file by hand; it is designed for that.

Note: `aicb init` does not write the sidecar, and neither does `init_solution_config` - the latter only gathers proposal material. The file is written by `Export Config`, by `apply_solution_config`, or by you.

Reading and writing rules

- Reading is all-or-nothing. A missing, unreadable or malformed file counts as "no sidecar file" - never as a partial configuration. A file whose `layerRules` is a string is a broken file, not "the layer axis is unset".
- The one deliberate exception is `suppressions`: it is read entry by entry, and a malformed entry costs only that entry. A suppression optimizes what a reader has to look at, while a broken layer or exclusion rule would silently change what the analysis reports, so those still fail the whole file.
- Comments and trailing commas are accepted, as are differently-cased key names and enum names. Numeric enum values are rejected, so `"layeringPolicy": "7"` does not silently become something.
- There are no key aliases. `layer_rules` is not `layerRules`: an unknown key configures no axis, and a file with no axis at all is not treated as a sidecar.
- A file with content on none of the six content axes (layer, exclusions, test, suppressions, analysis scope, build platform) loads as "no sidecar"; a file holding only `designTimeBuildPlatform` is a valid sidecar. The `autoInit` flags ride along with a content-bearing file; a file containing only flags is neither written nor read.
- Writing **replaces** the whole file, from one read of the file as it stands: axes the writer does not set and keys it does not know are carried over unchanged, and a file it cannot read - malformed, locked, or holding a keep rule inside `exclusions` - is not rewritten at all. Axes that are not present are omitted, so a file that uses only the original two axes stays byte-identical to the older format. `analyzePreferredTfmOnly` is written back exactly as it came in, including an explicit `false`, `designTimeBuildPlatform` is carried over unchanged, and the test attribute names keep the spelling the file has. No rule with an empty pattern or layer is written. The write is atomic (a temporary file of its own, then a replace), UTF-8, indented and camelCase; temporary files an interrupted write left are removed.

Which axis wins

The sidecar does not outrank everything - the precedence differs per axis:

Axis	What wins in the graphical interface	What wins in `aicb analyze`
Layer profile	explicit/per-solution database choice > global default > role-based heuristic. Sidecar rules participate only after they are restored into the database	`--layer-profile` > database (per solution > global default) > sidecar > role-based heuristic
Namespace exclusions	per-solution database choice > global default > built-in `BCL default`. Sidecar rules participate only after they are restored into the database	database (as soon as any database is available) > sidecar > built-in `BCL default`
Test profile	database (per solution > global) > built-in default	database (per solution > global) > built-in default - the sidecar's test axis is not part of this chain
Auto-init flags	database record, mirrored into the sidecar for a fresh clone	not evaluated
Suppressions	database, shared through the sidecar export	not evaluated
Analysis scope	sidecar only	sidecar only
Build platform	sidecar, else the built-in rule	sidecar, else the built-in rule (the MCP server follows the same order)

Note: if a database is available at all, the sidecar's exclusions are not consulted, even when the database list is empty. The sidecar is the database-free fallback.

Note: the analysis scope is not part of the file-set hash with which AICB decides whether a stored snapshot can be reused. `aicb analyze` therefore prints a note when a snapshot is reused although the scope may have changed since it was written, and points you to `--force`. The same holds for `designTimeBuildPlatform`: a snapshot holds the facts of the platform it was loaded at, and `aicb analyze` says so on a snapshot hit. When the sidecar names a platform, `aicb analyze` also prints it before the analysis, a refused value as a warning.

Note: after an edit of `designTimeBuildPlatform` alone, an unforced `refresh_session` keeps the platform the session was loaded at, because the `.aicb.json` is not a source file; `refresh_session(force: true)` always reloads the solution through MSBuild and applies the edited value. A new session, or a refresh that reloads the solution for another reason, reads it too.

7.4 How settings are resolved

Five separate resolution paths exist. They are independent of each other; confusing them is the fastest way to look in the wrong place.

Where settings live

Application settings come from two sources:

1. `app-settings.json` - it holds only three things: the storage settings (database path, base path, backup configuration), the recent solution paths and the recent database paths.
2. The configuration database - everything else, stored as key/value entries.

The split exists because the database path must be known before the database can be opened. The practical consequences:

- Every value that is neither in the file nor in the database falls back to the built-in default. The defaults have a single source, so a default is never defined twice.
- On the very first start, before the database schema exists, the built-in defaults are used and the settings are re-read once the schema is ready.
- A settings file that cannot be read does not crash the application: AICB continues with defaults and preserves a copy of the unreadable file next to it as `app-settings.json.corrupt-<timestamp>`. Because the file also holds the database path, the notice tells you that your sessions and solutions may look missing and where the preserved copy is.
- Note: the file always lives at `%APPDATA%\AIContextBuilder\app-settings.json`. The `App settings file` field under `Settings > Storage` is stored but does not move it.
- Note: settings that were removed in a later version stay in the database as inert entries. They are never read or written again, so a value you find there may simply do nothing.

Which profile is active

For the profile families, the first **resolvable, non-hidden** tier wins. A tier that is set but points at a hidden (soft-deleted) or deleted entry is skipped and the **next** tier applies - not the default. Only the last tier ignores the hidden marker, because it has to return something.

Profile	In the graphical interface	In MCP / headless
Pipeline profile	context template's `Pipeline profile` slot > globally active pipeline profile > built-in default	additionally the active MCP profile's template first

Profile	In the graphical interface	In MCP / headless
Quality profile	context template's `Insights profile` slot > globally active quality profile > built-in default	additionally the active MCP profile's template first
Test profile	per-solution test profile > globally active test profile > built-in default	identical
MCP profile	globally active MCP profile > built-in default (exactly one is active)	identical
Compression rules	globally active compression-rules profile > built-in default; a context template can override it through its slot	identical
Context template	globally active context template > `default`	the MCP profile's slot template for the facet > the MCP profile's template > globally active context template

Two consequences are worth knowing:

- Hiding an entry that an active context template still points at is silent: from the next render on the next tier applies, with no error and no log entry.
- On the shipped default configuration, changing the globally active quality, pipeline or compression profile in the graphical interface does **not** change MCP answers. All built-in MCP profiles point at the `AI Optimized` context template, and that template pins the built-in defaults for those three slots, so the template tier wins before your global setting is read. This is what pinning a template slot means, not a defect. What does move MCP answers is the corresponding slot of the template your active MCP profile names, or a different MCP profile.

The MCP server configuration file and environment variables

The MCP server has its own precedence:

```
built-in defaults  <  aicb.mcp.json  <  environment variable
```

`aicb.mcp.json` lives at `%APPDATA%\AIContextBuilder\aicb.mcp.json` and is only read, never written. If it is missing or unreadable it is ignored and the server starts on the built-in defaults. Its keys:

Key	Meaning
`toolSpec`	Which tools the server exposes: `"lean"`, `"all"`, or a comma-separated list of tool group names. Not set = the active profile's pool.
`sessionCache.ttlMinutes`	How long an analyzed session is kept, in minutes, sliding from the last access.
`sessionCache.maxSessions`	How many sessions are held at the same time.
`sessionCache.sweepMinutes`	Interval of the background cleanup, in minutes.

Environment variables override the file for the process they are set in:

Variable	Effect
`AICB_MCP_TOOLS`	Tool selection: `all` exposes every tool including the opt-in infrastructure tools; literal `lean` exposes the complete 72-tool lean core; a comma-separated list of tool group names narrows the selection; unset = no override, so the active profile's pool applies.
`AICB_MCP_SESSION`	Session cache, for example `ttlMinutes=120,maxSessions=4,sweepMinutes=15`.
`AICB_MCP_AUTO_REFRESH`	Overrides the automatic-refresh mode for this server process (`off` , `reactive` , `proactive`).
`AICB_MCP_ERROR_TEXT`	Set to `off` to record only the exception type in the server's usage report, not the message.
`AICB_MCP_DEBOUNCE_MS`	How long the source-change watcher waits before it reacts, in milliseconds.
`AICB_MCP_STALENESS_WINDOW_MS`	The window in which a session counts as stale, in milliseconds.
`AICB_MCP_LOAD_WAIT_MS`	How long a tool call waits at the solution load gate before it is told what is holding it, in milliseconds.
`APPDATA`	Determines the base path and therefore all storage locations.

For the three `*_MS` values, an unparsable or non-positive value falls back to the built-in default. Three further variables tune the analysis itself: `AICB_DESIGN_TIME_BUILD_CACHE` (`0` , `off` or `false` disables the design-time build cache; an absolute path relocates it), `AICB_ANALYZE_PREFETCH_DEPTH` (how many builds to keep in flight; `0` disables the prefetch) and `AICB_ANALYZE_NULLABLE_FLOW` (`1` , `true` , `yes` or `y` keeps nullable flow analysis in the semantic model).

The source badge on a field

Where a value can come from several tiers, a small pill next to the field shows where the effective value came from. In the `Sections` panel of the Context Builder it has three states, with the tooltip `Source of the active Tag Schema selection for this slot` :

Badge	Meaning
`default`	The built-in default.
`profile`	The value from the active MD profile.
`override`	A per-run override you made in this panel.

Behind this sits a general four-level ladder, from lowest to highest precedence: built-in code default → template slot → solution/session override → per-node override.

Token budget

Two resolution paths exist for a token budget:

- Template-aware MCP renders (`export_markdown` , `prepare_task`): explicit `budget` parameter > the active MCP profile's token budget > the context template's token budget > the global pipeline profile. No budget is enforced only when none of those sources supplies one and trimming is disabled; a trimming profile such as Aggressive Trimming can still impose its global budget.

- The slice tools (`get_context` , `explain_symbol` , `pack_for_task` and the other tools with a `budget` parameter): explicit `budget` parameter > the active MCP profile's token budget > a default ceiling of 10,000 tokens.

Both MCP paths floor the value at 8,000 tokens. The active MCP profile's token budget is the only setting both MCP paths honor. The graphical interface instead passes the current `Max MD size` slider value (or no cap when `No limit` is selected). The CLI has no explicit budget option; when rendering through a run template it uses that context template's pipeline-profile override, or the globally active pipeline profile when no override is set.

When a change takes effect

- Most settings take effect as soon as you save them. The library pages and the per-solution pickers stay in sync live.
- A change to `Base path` under `Settings > Storage` requires an application restart. The same holds for the LLM retry settings under `Settings > General > LLM Retry` .
- The MCP server reads the active MCP profile and the automatic-refresh mode once at start, so restart the server for a change to apply - the tool list and the server instructions are delivered when a client connects. The `AICB_MCP_AUTO_REFRESH` environment variable overrides the mode for one process. If the active profile changed since the server started, `server_info` reports the pending change. `aicb mcp --mcp-profile <id>` pins which profile a server process uses; the pin is process-local and writes nothing back.
- A session that has already been analyzed keeps the configuration it was analyzed with (see "Sessions and staleness"). `refresh_session` deliberately reuses the session's layer profile, exclusion list and analysis scope instead of re-reading the sidecar, so editing the sidecar does not change a running session. Analyze the solution again, or use `refresh_remembered` , which returns a new session and reads the sidecar again.
- The run-confirmation threshold under `Settings > General > Run Confirmation` is read at each run, so a change applies immediately. Note: the value `0` means that **every** run asks for confirmation; there is no "never ask" value. Enter a very high number if you want to switch the pre-flight confirmation off.

8 Insights: the code quality catalog

AICB analyzes your solution and reports what it finds as **insights**: short, contextual hints about your C# code - a long method, an unused type, a broken XAML binding, a dependency that points the wrong way. Each insight names the affected code, explains what the rule looks for, suggests a fix, and estimates the remediation effort. This chapter describes where insights appear, every rule in the catalog, the quality profiles that control them, how to mark findings as reviewed, and the limits the analysis has by design.

8.1 What an insight is

An insight is produced by a **rule** (internally called a producer): one small heuristic that walks the analyzed solution and emits a finding when its condition matches. Every insight is a self-contained report with the same shape:

Element	Meaning
Title	Short summary; usually carries the number of hits and the rule's unit (e.g. "12 methods over 50 LOC").
Description	What the rule detected, why it matters, and any caveat specific to the rule.
`Fix:` line	A concrete, actionable suggestion. Some insights have none.
Detail lines	One line per affected code location, in a monospace font. A line that resolves to a symbol carries a **Reveal in the Solution Tree** button and can be selected, queued, suppressed, or dismissed individually.
Severity	`Info`, `Ok`, `Warning` or `Critical` - see "Severity levels".
Category	Where the card is grouped in the Insights tab - see "Categories".
SOLID badges	The design principle(s) the rule addresses (`S`, `O`, `L`, `I`, `D`, or `·` for "Other"). Hover a badge to read what the letter stands for.
Remediation cost	An order-of-magnitude estimate such as `~15 min`, when the rule can estimate one. It feeds the technical-debt rating.

Findings are grouped per rule, not per code location: one insight aggregates every hit of its rule (for example all anti-pattern hits in the solution). The individual locations are the detail lines.

Running the analysis

Insights are not computed while you type, and a solution load does not block on them:

1. Open a solution. AICB restores the last persisted insight run for that solution (when `Persist insights with session` is on) and starts a fresh analysis in the background. The fresh result replaces the restored one as soon as it is ready, so what you see matches the current state of the tree.
2. To re-evaluate on demand, click **Analyze Solution** in the Insights tab header, or press `Ctrl+Shift+A` from any sub-tab. The button is the single accent action of the tab; it runs all rules over the active solution.
3. On a fresh installation, the rules run once automatically on the first solution load so the tab is not empty. Afterwards the on-demand behavior applies.
4. The optional setting `Auto-trigger insights on solution load` (Settings → General, section "Session & Analysis") starts the run directly as part of every solution load instead of the background preparation. `Persist insights with session` controls whether results are stored per solution and restored on the next load; with it off, insights are recomputed fresh every time and never written to the database.

A full run takes roughly 3-15 seconds depending on solution size. During the run the tab is dimmed by a spinner overlay; the analysis works on a read-only snapshot and cannot disturb your editing session.

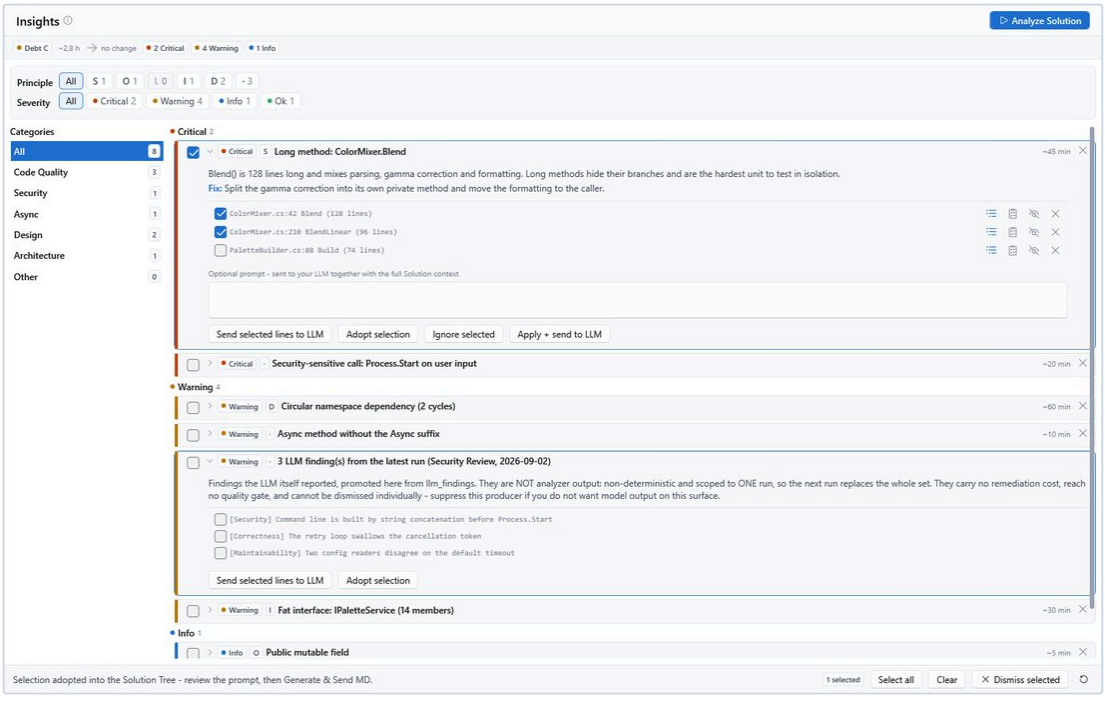
If no solution of your own is loaded yet, the empty state offers **Open sample Solution**, which loads the bundled read-only sample `ColorMixer.SelectionLab.sln` for a trial run.

The Insights tab

The tab has four regions:

- A **page header** with the `Insights` title, an info tooltip, and the `Analyze Solution` button.
- A **metrics strip** over all findings (it appears once a run has produced results): the technical-debt grade `Debt A` … `Debt E`, the estimated total remediation time (e.g. `~3.5 h`), a trend chip comparing the stamped debt with the previous analysis (`Worsened`, `Improved`, or no change; shown only when at least two stamped snapshots exist), and the counts `N Critical`, `N Warning`, `N Info`.
- Two **filter rows**, both multi-select toggle sets:
- `Principle`: `All`, `S`, `O`, `L`, `I`, `D`, and `·` for "Other". Each chip shows how many findings it currently holds; a chip with count 0 is not clickable. `All` clears the axis.
- `Severity`: `All`, `Critical`, `Warning`, `Info`, `Ok`. Several severities can be active at once.
- A **two-pane body**: a single-select category navigator on the left (with a running count per category) and the selected category's cards on the right, grouped under severity headers (`Critical`, `Warning`, `Info`, `Ok`) with a count each.

If the right pane is empty, it says: `No insights to show here - pick a category on the left, or relax the principle/severity filters.`



The Insights tab: categories on the left, findings grouped by severity on the right

Working with a card

A card starts collapsed as a one-line row: severity pill, SOLID badges, title, and the remediation cost. Click the row (or the chevron) to expand it into the description, the `Fix:` line, and the detail list. The list is height-capped and scrollable; the Insights tab shows every hit line.

- The checkbox at the left of a card selects it for a batch action (`Select for a batch action (Dismiss selected).`).
- The **Dismiss** button at the top right marks the whole card as seen (`Dismiss - marks the finding as seen; it will not come back`). It is hidden on findings that cannot be dismissed.
- With the card expanded, each detail line has its own checkbox and up to five small buttons:
- **Reveal in Solution Tree** - selects and reveals the line's symbol in the tree.
- **Queue as work** - opens a picker asking for a `Run template` , then records the line as work to do (`Queue`). Nothing is started; the line stays visible and reports `In progress` .
- **Mark done** - closes the queued item. This records your claim; nothing re-scans to confirm it, and the line stays visible so a later run can contradict it. A closed line reports `Done` .
- **Intentional here** - records a suppression scoped to the line's symbol. The tooltip is explicit about the scope: `Mark this finding as intentional - it stops being reported here and to the MCP. The quality gate (--fail-on / solution_metrics) still counts it. Export the solution config to share the decision via the .aicb.json sidecar.`
- **Seen this one** - dismisses just this line, locally (`Dismiss just this line - it stops being reported here and to the MCP until you restore dismissed findings.`). It appears only when the line resolves to a scope narrower than the whole rule; to silence a whole rule deliberately, use the card-level Dismiss.
- Below the detail list, an optional prompt box appears (`Optional prompt - sent to your LLM together with the full Solution context`), and up to three action buttons:
- **Send selected lines to LLM** - sends the checked detail lines, together with the full solution context, to the active LLM.
- **Adopt selection** - selects the checked symbols (and pulls in their relevant neighbors) in the Solution Tree, replacing the current selection, and writes a fix task into the prompt. Review it, then use `Generate & Send MD` .
- **Ignore selected** - dismisses the whole card, the same as the top-right Dismiss button.
- The insight-card model has support for a rule-specific secondary action, but the only producer that supplies one is not registered in the released application. No shipped card currently shows that button.

The action bar at the bottom of the tab holds the batch actions: a selection summary pill, `Select all` , `Clear` , `Dismiss selected` (marks all checked findings as seen), and an icon button whose tooltip reads: `Reset what you hid - dismissed Insights and findings you marked intentional become visible again, and all sections are re-evaluated immediately. Suppressions declared in the committed .aicb.json sidecar stay in force.`

Note: A card's title count and its cost estimate are computed from the full hit list *before* triage. On a card whose lines you have partly hidden, the title can therefore still name the original total while the list shows only the remainder. The MCP answer carries a `suppressed` field reporting how many lines were removed by triage.

Note: The Insights tab lists every hit line. `list_insights` returns summary fields, including `detailsCount` , but no detail lines - unless it is called with `files` , which returns the detail lines about the types those files declare (20 per insight, then `(... and N more)`). `get_insight` returns detail lines, caps them at 20 by default, appends `(... and N more)` when truncated, and accepts `maxDetails` to raise the cap for a drill-down.

8.2 Severity levels

Every insight carries one of four severities. They are used for the card ribbon, the grouping headers, the filter chips, the header counts, and the quality gate.

Severity	Meaning	Weight used for sorting and scoring
`Critical`	A structural problem that should be addressed before release (e.g. a namespace cycle spanning assemblies).	10
`Warning`	A defect or convention break with a practical consequence (broken binding, resource leak, blocking async call).	3
`Info`	A heuristic hint or refactoring suggestion. The majority of the catalog.	0.5
`Ok`	Neutral/no action needed. Present in the model and offered as a filter chip; no built-in rule currently emits it.	0

The weights are deliberately not proportional to the order of the levels; they are what the debt and gate scoring use. On the wire and in persisted data the levels travel as the lowercase tokens `info`, `ok`, `warning`, `critical`; unknown or legacy tokens read back as `Info` rather than failing.

8.3 Categories

Each rule belongs to exactly one category, which decides where its card is sorted in the left navigator. The navigator always lists `All` plus `Code Quality`, `Security`, `Async`, `Design`, `Architecture`, and `Other`; further categories appear only when a finding actually maps to them.

Category	What it collects
`Code Quality`	Quality heuristics, correctness smells, dead-code suspicion, complexity, size and parameter checks, side-effect concentration, deprecated APIs, resource leaks, reflection and LINQ-in-loop usage.
`Design`	Design weaknesses: unused types, public mutable fields, many parameters, layer violations, concentrated event subscriptions, unresolved XAML bindings.
`Async`	.NET async-convention checks: missing `Async` suffix, missing `CancellationToken`.
`Architecture`	Circular namespace dependencies and public-contract breaking changes.
`Security`	Call sites of security-sensitive external APIs.
`Other`	Findings without a category mapping (currently used as a fallback only).
`Compression`	Reserved for the auto-compression tips; no active rule reports into it (see "Not yet released").

The category is independent of the SOLID badge: the category says where a card is sorted, the badge says which design rule it addresses.

8.4 The insight catalog

The catalog lists 27 entries: 24 analyzer rules, the LLM findings, and one pair of auto-compression tips that is not yet released (see "Not yet released").

Insight id	Category	Severity	Enabled by	What it reports
`quality-long-methods`	Code Quality	Info	`Long methods (LOC over threshold)`	Methods longer than the LOC threshold, or - when the complexity option is on - methods whose cyclomatic complexity exceeds the complexity threshold.
`quality-dead-code-private`	Code Quality	Warning	`Dead private methods (UsedBy == 0)`	Private methods with no detected caller.
`quality-fat-interfaces`	Code Quality	Info	`Fat interfaces (methods over threshold)`	Interfaces with more methods than the threshold (suspected ISP violation).
`quality-large-classes`	Code Quality	Info	`Large classes (members over threshold)`	Classes with more members (methods + properties + fields) than the threshold (suspected SRP violation).
`quality-anti-patterns`	Code Quality	Warning	`Anti-patterns (correctness smells: ...)`	Correctness and convention smells: blocking on async (`.Result` / `.Wait`), direct clock reads, `new HttpClient()` , `async void` , dropped fire-and-forget tasks, broad `catch (Exception)` , `Thread.Sleep` , missing `ConfigureAwait(false)` , and a constant `new Regex()` that could be `[GeneratedRegex]` .
`quality-complexity-hotspots`	Code Quality	Info	`Complexity hotspots (top methods by cognitive complexity)`	Methods ranked by cognitive complexity, worst first.
`quality-complex-untested`	Code Quality	Warning	`Complexity hotspots`	Complex methods with no direct test, from the intersection of the complexity ranking and the coverage-gap analysis.
`quality-multi-concern-hotspots`	Code Quality	Info	`Complexity hotspots` + `Side-effect concentration` + `Anti-patterns`	Methods flagged by two or more quality dimensions at once - the highest-priority refactor targets.
`quality-side-effect-concentration`	Code Quality	Info	`Side-effect concentration (methods mixing 3+ effect categories)`	Methods mixing three or more distinct side-effect categories (I/O, network, database, serialization, logging, cache, messaging).
`quality-linq-in-loop`	Code Quality	Info	`Anti-patterns`	Methods evaluating a LINQ operator inside a loop body.
`quality-reflection-usage`	Code Quality	Info	`Anti-patterns`	Methods whose resolved external calls or reads name reflection infrastructure.

Insight id	Category	Severity	Enabled by	What it reports
`quality-resource-leaks`	Code Quality	Warning	`Anti-patterns`	Disposables created with `new` whose ownership is neither transferred nor disposed.
`quality-deprecated-referenced`	Code Quality	Info	`Anti-patterns`	Types or methods marked `[Obsolete]` that still have an intra-solution reference.
`quality-event-subscription-concentration`	Design	Info	`Side-effect concentration`	Types subscribing to three or more events without a matching unsubscribe.
`design-unused-type`	Design	Info	`Unused types (no inbound usage)`	Types with no inbound reference, after the structural exemptions described under "Known limits of the analysis".
`design-public-mutable-field`	Design	Warning	`Public mutable fields (encapsulation break)`	Public fields without `readonly` / `const`.
`design-many-parameters`	Design	Info	`Many parameters (over threshold)`	Methods with more parameters than the threshold. Constructors are excluded.
`quality-layer-violations`	Design	Warning (Info for the coverage note; `Critical` under a Strict layer policy)	`Layer violations (inner layer depends on outer layer)`	Dependencies that contradict the Clean/Onion rule: a type in an inner layer depends on a type in an outer layer.
`design-unresolved-xaml-binding`	Design	Warning	`Unresolved XAML bindings`	WPF/Avalonia `{Binding X}` bindings whose root member does not exist on the confidently resolved DataContext view-model.
`async-missing-suffix`	Async	Warning	`Missing Async suffix on Task-/ValueTask-returning methods`	Task/ValueTask methods whose name does not end in `Async`.
`async-no-cancellation-token`	Async	Warning	`Async method without CancellationToken parameter`	Task/ValueTask methods without a `CancellationToken` parameter.
`quality-circular-dependencies`	Architecture	Warning; `Critical` when a cycle spans several assemblies	`Circular namespace dependencies (dependency cycles)`	Namespaces that reference each other in a cycle (`A → B → A` or a larger tangle).

Insight id	Category	Severity	Enabled by	What it reports
`quality-breaking-changes`	Architecture	Warning	`Circular namespace dependencies`	Public API removals or signature changes compared with the latest saved snapshot. Needs a saved snapshot; without a baseline there is no finding.
`security-sensitive-calls`	Security	Warning	`Anti-patterns`	Methods calling security-sensitive external APIs: insecure deserializers (`BinaryFormatter` , `SoapFormatter` , `NetDataContractSerializer` , `LosFormatter`), `Process.Start` , or `Assembly.Load` .
`llm-findings`	Code Quality	Taken from the category the model itself declared	- (always on)	Findings the LLM reported in the latest run, promoted from the run's findings.
`autocompression-firstuse`	Compression	Info	-	Not yet released; see below.
`autocompression-reminder`	Compression	Info	-	Not yet released; see below.

Notes on individual rules

- **Long methods.** LOC counts only the lines that carry code - blank and comment-only lines are not counted, so documenting a method does not make it "long". By default the rule *also* sharpens on cyclomatic complexity, so a short but heavily branched method is flagged too. Both thresholds are editable in the quality profile.
- **Anti-patterns.** The rule aggregates every hit of its kind into one card and shows the per-kind distribution in the description; the detail list is sorted alphabetically by `Type.Method` , so a capped list shows the first entries, not a sample per kind. The convention smells (broad catch, `Thread.Sleep` , `ConfigureAwait` , `Regex`) are recall-safe - they also work on a restored session. The remaining smells are detected on live sessions only.
- **Complex, untested.** "Untested" here is a one-hop index: a method that no test invokes *directly* is listed even when a test drives it through intermediate calls. Entries marked as reaching no test at all are listed first, and only those count towards the cost estimate. The rule is a prioritization signal, not a coverage measurement; for a real coverage view use `coverage_gaps` .
- **Complexity hotspots** ranks by *cognitive* complexity (nesting and comprehension load), using the same threshold number as the cyclomatic axis. It complements the long-method rule, which thresholds cyclomatically, so the two do not double-report.
- **Multi-concern hotspots** is a prioritization ranking over the other rules, not a separate finding: it counts a method once per dimension that flags it (complexity, side-effect concentration, reflection, concurrency, resource leaks, security-sensitive calls), and only dimensions whose switches are on are counted. It deliberately carries no remediation cost, because the underlying concerns already carry theirs.
- **Side-effect concentration** counts *real* effect categories; an unanalyzable external call is not counted as a category, so it cannot inflate the number.
- **Event-subscription concentration** lists types that subscribe to three or more C# events with `+=` and declare no matching `-=` . An event-heavy type is a coupling smell and a lifetime risk; types that release everything they wire are not listed.

- **LINQ in a loop** is detected strictly: only real `System.Linq` operators count (not a same-named instance method such as `HashSet.Contains`), and only inside the loop body - a `foreach` source runs once and is not flagged. The finding is often benign on a small local collection.
- **Reflection usage** matches on the receiver type (`System.Reflection.*`, `Type.*`, `Activator`, `AppDomain`, `object.GetType()`, expression trees), never on a bare member name, so a `DependencyObject.GetValue` accessor or a `typeof(...)` in an attribute does not count. It is informational: much reflection is intentional, but it defeats the IL trimmer and can break under NativeAOT.
- **Resource leaks** is deliberately conservative: it reports only a bare `new X();` statement or a local that is never referenced again, so a false positive is effectively ruled out. Each hit is estimated at 15 minutes. A leaked disposable is treated as a correctness smell like the other anti-patterns, which is why it shares their switch.
- **Deprecated APIs still referenced** omits deprecated symbols with no detected reference (those are safe to delete, not debt). Two limits: "still referenced" uses the reverse type fan-in for types and the caller list for methods, so a symbol reached only via reflection or DI shows no reference; and the type fan-in is keyed by simple name, so two solution types sharing a name share one count. Only the deprecated side is restricted to production code - a reference from a test project still counts.
- **Layer violations.** The rule resolves layers from namespace conventions plus a role heuristic, and the severity follows the layer profile's policy: `Advisory` → `Warning`, `Strict` → `Critical`. When the check could not classify every production type, the description names how many types were checked and which were skipped, so a clean result over a partly unchecked solution is not mistaken for a clean solution.
- **Circular namespace dependencies.** Granularity is the namespace (type-level cycles are usually benign in C#). A cycle that spans assemblies is the more serious case and raises the severity to `Critical`. The detail line anchors on a concrete witness type so `Reveal in Solution Tree` and `Adopt selection` work.
- **Public-contract breaking changes.** Compares the current public/protected surface against the latest saved snapshot with identical options. A removed public type or member, or a changed member signature, is breaking; a pure accessibility widening is conservatively reported as a change too (over-warning is the safe direction). Additive changes are not listed. Save a snapshot (see the Sessions and snapshots documentation) to establish a baseline.
- **Unresolved XAML bindings.** The compiler does not catch this class of defect: a binding to a member that does not exist fails silently at runtime. Only view-models resolved with certainty are checked (WPF: `d:DesignInstance` or a unique `FooView` → `FooViewModel` convention; Avalonia `.axaml` : `x:DataType`), and only when their member surface is verifiable. Source-generated members (`[ObservableProperty]`, `[RelayCommand]`) and inherited members count as resolved; a view-model with an unknown external base is skipped, not flagged. Live-only: re-run the analysis after edits.
- **LLM findings** are not analyzer output. They are parsed from the latest run's output, are non-deterministic, and are scoped to that one run: the next run replaces the whole set and may not repeat any of it. A multi-step run contributes the findings of every reasoning step, so an earlier step can contradict a later one. They carry no remediation cost and cannot be dismissed.

Not yet released

The auto-compression tips (`autocompression-firstuse`, `autocompression-reminder`, category `Compression`) are planned and not yet released: they do not appear in the Insights tab or in an MCP answer, and the category `Compression` stays empty until they are.

8.5 Quality profiles

A **quality profile** is a bundle of rule switches and thresholds. It decides which rules run, how strict their limits are, and how their actions behave.

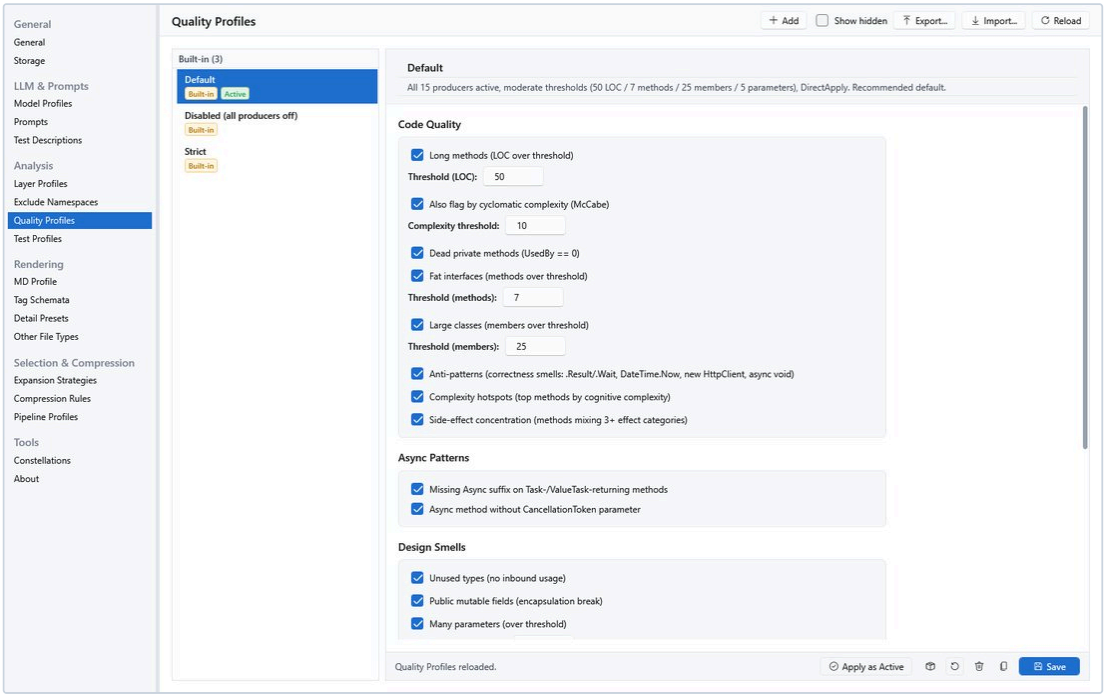
The three built-in profiles

Id	Name	Description as shown in the picker
`quality-profile/default`	`Default`	`All 15 producers active, moderate thresholds (50 LOC / 7 methods / 25 members / 5 parameters), DirectApply. Recommended default.`
`quality-profile/strict`	`Strict`	`Stricter thresholds (30 LOC / 5 methods / 15 members / 3 parameters), ConfirmDialog. Suited to code-review sessions on legacy code.`
`quality-profile/disabled`	`Disabled (all producers off)`	`All 15 producers turned off. For users who want to disable findings entirely without hiding the tab.`

Note: The profile model contains 15 producer switches, but the desktop editor exposes only 14 of them; `Unresolved XAML bindings` is currently not editable in the UI. The editor's fifteenth checkbox is the separate McCabe option. The catalog contains 24 analyzer rules, and several producer switches control more than one rule (see the table below).

Selecting and editing a profile

Open **Settings** → **Quality Profiles**. The panel is a master-detail editor:



The Quality Profiles page: producer switches and thresholds

- The list on the left shows the built-in and custom profiles; the detail pane shows the selected profile's name, description, and all settings.
- **Apply as Active** makes the selected profile the global default: `Apply as Active: make this profile the global default. Every insight heuristic uses it unless a template overrides it.`
- The active profile is resolved in this order: the active context template's quality-profile slot (when set) → the globally active profile → the `Default` built-in. A profile chosen on a run template therefore takes precedence over the global choice.

- Built-ins can be edited; saving marks the built-in as overridden. **Restore Built-In** resets it to its code defaults and clears the override. **Duplicate** creates a custom copy to tweak; **Delete** hides a built-in (recoverable via Restore) or removes a custom profile permanently. **Export as Built-In** shows the entry as a C# snippet; it is intended for the product's own development and has no effect on your installation.
- `Save` persists editor changes (`Ctrl+S`), `Duplicate` is `Ctrl+D` .

Producer switches and thresholds

The profile model contains 15 producer switches and 5 numeric thresholds. The desktop editor exposes 14 producer switches (all rows below except `Unresolved XAML bindings`) plus its separate McCabe checkbox. Defaults are shown in the tables; the `Strict` profile changes the numeric values as noted.

Switch	Default	Rules it controls
`Long methods (LOC over threshold)`	on	`quality-long-methods`
`Dead private methods (UsedBy == 0)`	on	`quality-dead-code-private`
`Fat interfaces (methods over threshold)`	on	`quality-fat-interfaces`
`Large classes (members over threshold)`	on	`quality-large-classes`
`Anti-patterns (correctness smells: .Result/.Wait, DateTime.Now, new HttpClient, async void)`	on	`quality-anti-patterns` , `quality-deprecated-referenced` , `quality-linq-in-loop` , `quality-reflection-usage` , `quality-resource-leaks` , `security-sensitive-calls`
`Complexity hotspots (top methods by cognitive complexity)`	on	`quality-complexity-hotspots` , `quality-complex-untested` , `quality-multi-concern-hotspots`
`Side-effect concentration (methods mixing 3+ effect categories)`	on	`quality-side-effect-concentration` , `quality-event-subscription-concentration`
`Missing Async suffix on Task-/ValueTask-returning methods`	on	`async-missing-suffix`
`Async method without CancellationToken parameter`	on	`async-no-cancellation-token`
`Unused types (no inbound usage)`	on	`design-unused-type`
`Public mutable fields (encapsulation break)`	on	`design-public-mutable-field`
`Many parameters (over threshold)`	on	`design-many-parameters`
`Layer violations (inner layer depends on outer layer)`	on	`quality-layer-violations`
`Circular namespace dependencies (dependency cycles)`	on	`quality-circular-dependencies` , `quality-breaking-changes`
`Unresolved XAML bindings`	on	`design-unresolved-xaml-binding`

Threshold	Default	Strict	Used by
`Threshold (LOC):`	50	30	Long methods
`Complexity threshold:`	10	7	Long-method complexity sharpening, complexity hotspots, complex-untested, multi-concern hotspots
`Threshold (methods):`	7	5	Fat interfaces
`Threshold (members):`	25	15	Large classes
`Threshold (parameters):`	5	3	Many parameters

Note: Some switches are broader than their label suggests. `Anti-patterns` also controls the security-sensitive-call rule and the deprecated-API rule; `Circular namespace dependencies` also controls the public-contract breaking-change rule. If you switch one of these off, the coupled rules stop reporting as well. Two rules (`llm-findings` and the not-yet-released auto-compression tips) have no switch at all.

Behaviour settings

Setting	Meaning
`Insights are dismissable (persist hidden per solution)`	Default on. When on, you can dismiss individual insights per solution. When off, all insights are persistent and come back after dismissing until the rule itself stops reporting them.
`Send-Template-Id override (optional)`	Stored support for the rule-specific `Apply + send to LLM` path. Empty uses the active tab's run template; an unknown id falls back to it. The only producer that supplies this secondary action is not registered in the released application, so the setting currently has no visible card to act on.
`Action mode`	Stored mode for that same unreleased secondary-action path: `DirectApply` or `ConfirmDialog` . It currently has no effect on shipped insight cards.

8.6 Suppressing and dismissing findings

Triage has two separate axes, and the distinction matters when you share your decisions with a team:

	Intentional (suppression)	**Seen** (dismissal)
Means	"This hit is by design."	"I have read this."
Scope	A member, a type, or a whole rule.	A member, a type, or a whole rule.
Stored in	The database, and - once exported - the git-tracked `<Solution>.aicb.json` sidecar next to your solution file.	The solution record in the local database, not in the sidecar.
Shared with the team	Yes, through the sidecar.	No, it is local.

Both axes are applied on the **reading surfaces**: the Insights tab and the MCP tools `list_insights` and `get_insight` . The quality gate is deliberately *not* filtered - see "What the triage affects".

Silencing a single line

1. Run `Analyze Solution` and expand the card.
2. To record a decision the team should see, click **Intentional here** on the line. AICB records a suppression scoped to that line's symbol.
3. To simply stop seeing the line locally, click **Seen this one**. AICB records a dismissal scoped to that line's symbol.
4. Export the solution configuration from the Solutions tab to write the suppressions into the `.aicb.json` sidecar. Commit the file so the decision travels with the repository; entries declared there stay in force even when you reset your local triage.

Lines that carry no resolvable symbol (relation or marker lines) cannot be suppressed or dismissed individually; there is nothing stable to key them on.

Silencing a whole rule or a whole type

- **Whole card / whole rule:** click the Dismiss button at the top right of the card, or check the card and use `Dismiss selected` in the action bar, or expand the card and click `Ignore selected`.
- **A whole type:** suppress or dismiss a line whose scope covers the type; the recorded scope is as narrow as the line allows. Silencing a whole rule from a per-line button is deliberately not offered, because one click must not claim every hit of the rule.

Restoring what you hid

The icon button in the action bar resets what you hid: dismissed insights and findings you marked intentional become visible again, and all sections are re-evaluated immediately. Suppressions declared in the committed `.aicb.json` sidecar stay in force - they are a team decision, not a local one, and are not dropped by a panel button.

Findings that are not dismissable (for example `llm-findings`, or every insight while `Insights are dismissable` is off) show no Dismiss button. Such a card stays visible until the rule itself stops reporting it.

What the triage affects

- **Affected:** the Insights tab, and the MCP tools `list_insights` / `get_insight`. An insight whose detail lines are all filtered out does not appear at all. A partially filtered insight reports the number of removed lines as `suppressed` in `list_insights` and as `suppressedDetailCount` in `get_insight`; that number covers both axes together and is not a count of sidecar entries.
- **Not affected:** the quality gate - the CLI `aicb analyze --fail-on "<expression>"` and the MCP `solution_metrics` rollup. A gate that the gated party can silence is a weaker gate, so suppressions and dismissals never flip a gate result. The gate aggregates the analyzer rules only; it does not include the LLM findings or the snapshot-based public-contract check.

8.7 The technical debt estimate

Insights that can estimate their remediation effort carry a `~N min` label. The estimates are deliberately coarse (order of magnitude, not precision). The tab sums them into a solution-wide debt and maps it onto a coarse A-E grade.

Rule	Estimate per hit
Long method	5 min + 1 min per 10 LOC over the threshold

Rule	Estimate per hit
Fat interface	10 min + 3 min per method over the threshold
Large class	10 min + 1 min per 5 members over the threshold
Many parameters	10 min
Dead private method	5 min
Unused type	10 min
Public mutable field	10 min
Missing `Async` suffix	5 min
Async without `CancellationToken`	5 min
Layer violation	20 min
Circular namespace dependency	20 min
Side-effect concentration	12 min
Unresolved XAML binding	8 min
Complex but untested	8 min + 1 min per cyclomatic-complexity point
Deprecated API still referenced	10 min
Public-contract breaking change	15 min
Anti-pattern hit	12 min
Resource leak	15 min
Security-sensitive call	15 min per method
Event-subscription concentration, complexity hotspots, multi-concern hotspots, LINQ in a loop, reflection usage, LLM findings	no estimate (rankings and informational findings)

Rating	Summed remediation time
`A`	under 30 minutes - very low debt
`B`	from 30 minutes
`C`	from 2 hours
`D`	from 1 working day (8 hours)
`E`	from 3 working days (24 hours) - high debt

The header shows the estimate in coarse buckets (`min` , `h` , or working days) and never as an exact number; treat it as an order of magnitude.

8.8 Known limits of the analysis

The analysis is heuristic. This section states what it does not see, so you can judge a finding instead of trusting or distrusting it blindly.

How the analysis errs

The guiding rule is **under-report rather than over-report**: a missed finding is cheaper than a false alarm in a tool you use for trust and deletion decisions. The one deliberate exception is input validation: a misspelled token that would otherwise produce a plausible but wrong result is rejected outright rather than silently treated as a lenient default.

Structural exemptions

Some code is alive but structurally invisible to the signal that looks for inbound references. AICB recognizes four families and excludes them from the unused-type check rather than reporting a false positive. The public-mutable-field check has its own narrower exemptions: structs and runner-activated types.

Family	Why the code is invisible	Examples
Kind or shape	Interfaces are resolved polymorphically via DI; a static class is reached only as `TypeName.Member`, and a member access is not a type reference.	Interfaces, static classes, `*Extensions` containers.
Test or benchmark scaffolding	Instantiated reflectively by a test or benchmark runner, or living in test infrastructure.	Test projects, `*Tests` / `*Fixture` names, test/benchmark attributes.
Framework-, reflection- or DI-activated	Plugged into an abstraction or a host convention and instantiated by the framework.	EF migrations, ASP.NET Core middleware and MVC controllers, entry points with `static Main`, Blazor pages, source-generator hosts.
Markup-referenced	Referenced from XAML rather than from C#.	Code-behind types, converters, `*View` / `*Window` / `*Dialog` / `*Page` naming, `*.xaml.cs` files.

A remaining caveat is stated with the finding itself: further reflection or plug-in patterns may still produce a false positive, so cross-check an unused-type hit with `find_usages` before deleting.

Heuristic calibration

- **Name conventions** are the basis for several role, layer and return-value classifications. They are matched on structural patterns (for example `On<Upper>` for event handlers, `*Result` for result types), not on bare substrings, so ordinary words do not latch.
- **Pure versus impure** types in effect namespaces are carved out: types such as `Path`, `MemoryStream`, `StringReader`, `StringWriter` and the value/message types of the network namespace are treated as benign, so a pure helper in `System.IO` or `System.Net` is not reported as a side effect. A mis-classified external symbol would propagate through the call graph, which is why these carve-outs are precision-critical.
- **Similarity anchors:** similarity tools require a real structural anchor (a base type or a non-ubiquitous interface) and ignore ubiquitous ones such as `IEquatable`; otherwise generic skeletons would flood the result.

- **Production focus versus discovery.** Several aggregate and orientation tools hide test projects by default and offer an opt-in `includeTests` parameter - this applies to `find_by_side_effects`, the `symbol_metrics` ranking, `architecture_overview`, `find_dead_code` and `calls_external`. `coverage_gaps` has no `includeTests` parameter and always excludes test projects; the quality-rule reading tools likewise expose no such parameter. Discovering tools deliberately do *not* hide tests by default: `find_by_attribute` (for example with `Fact`), `find_usages` and `instantiation_sites` want to see test code. `find_usages` can be asked for production users only (`includeTests: false`), which also keeps test users from filling its 100-item cap. Interpret each tool according to its own signature rather than assuming a shared switch.
- **Namespace normalization** strips `global::` qualifiers and separates framework/generated namespaces from your own architecture, so configuration tools compare the right namespace sets.
- **Layer inference from project names** lets an outermost-ring host segment dominate an inner layer word: a project named like a web/api/cli/mcp/composition/benchmark host is classified as `Presentation` even when its name also contains `Application` or `Domain`. This is correct for a host and a known false positive when "Api" means the public surface of a library.

Input validation

- **Did-you-mean on empty:** when a token produces no hits, the answer lists the values that actually exist in the scope - for example `availableEffects`, `availableReturnsSemantics`, or `availableAxisValues`. A typo therefore returns a self-explaining empty answer rather than a bare zero. The successful path is unchanged.
- **Strict rejection:** when a wrong token would fall into a populated but wrong result, the tool rejects it at the boundary with an error instead of silently degrading to a more lenient value. This applies, for example, to an unknown policy value, an unknown `matchType`, or an unknown `minConfidence` / `purity` token.

Limits you should know

- **Caller lists are statically resolved.** Reflection, DI containers, serializers, and external assemblies stay invisible. The absence of callers is not proof of dead code. The compression legend in every generated document states this, and it applies to the insights as well.
- **Resolution completeness is not the same as a successful restore.** A run reports its projects as resolved when they bound their core references; a project that bound `System.Object` from the base framework but is missing its own package assets still counts as resolved, so treat "all resolved" as a statement about a wholly unbuilt tree, not as a build guarantee.
- **Afferent coupling (Ca) is 0 on live analyses.** The live surfaces (the gate's `ca-max`, the `QUALITY_HOTSPOTS` section, the `ca=""` tag) compute the reverse fan-in from the dependency data instead, so the displayed value is meaningful; only a stored field in hand-built models and older snapshots carries 0.
- **Effect, leak and external-call statements need a built tree.** On an unbuilt working tree, fan-in statements can partly survive, but effect statements do not - `find_by_side_effects` can report 0 effects for a solution whose built form has hundreds. Build (or restore) the solution before trusting an effect answer.
- **Two complexity axes, two result sets.** The default ranking uses cyclomatic complexity; ranking by cognitive complexity can produce a different set of methods, not just a different order. `symbol_metrics` discloses this when the sets differ, not when the order merely changes.
- **Two complexity thresholds, two boundaries.** The solution rollup counts methods *strictly above* the threshold; `symbol_metrics`' `minComplexity` filter and the complex-untested insight include methods *at* the threshold. At the same number, the rollup's count is therefore smaller by exactly the methods on the boundary.
- **The layer check has two deliberate blind spots.** Dependency strings that name a generic (for example `IRepository<Order>` against the index key `IRepository`) are not matched, and edges inside one assembly without an authoritative layer assignment are not checked. The coverage note on the insight names how

many types were checked.

- **The XAML binding check skips silently.** A binding whose DataContext cannot be typed with certainty - a nested runtime `DataContext`, a template without a `DataType`, a `ControlTemplate` or `Style` - is never checked and never reported. An empty `find_unresolved_bindings` result therefore means "nothing broken among the bindings that could be checked", not "all bindings are good".
- **Snapshot-based checks need a snapshot.** The public-contract breaking-change rule compares against the latest saved snapshot; without one it reports nothing. Save a snapshot before a refactoring wave to get a meaningful baseline.
- **LLM findings are non-deterministic.** They are scoped to one run and replaced wholesale by the next run; they are not a trend and not a stable baseline.

9 Command-line reference

`aicb` is the command-line interface of AIContextBuilder. It runs the same analysis engine as the desktop application, but writes its result to a file, reports success and failure through exit codes, and can therefore be used from scripts, build servers and other tools. It is also the way to start the MCP server and to invoke a single MCP tool without a client.

The analysis verb `analyze` needs MSBuild on the machine - a .NET SDK or Visual Studio provides it. The other verbs work without it. Self-contained publishing removes the .NET runtime dependency, not the MSBuild one.

9.1 Invocation, help and version

```
aicb <verb> [options]
```

Verb	Purpose
`init`	Wire `aicb` into a project for an MCP client: the `.mcp.json` server entry, the agent skills and the symbol guard.
`analyze`	Run the Roslyn analysis on a solution and write the context document.
`export`	Re-render the context document from an existing session database, without a new analysis.
`import`	Import a constellation JSON (templates, presets, profiles, app settings) into a database.
`list`	List built-in and custom master data.
`mcp`	Start the MCP server over stdio.
`call`	Invoke exactly one MCP tool once and print its result.

Every verb accepts `--help` (also `-h` and `-?`), which prints its options and their descriptions. `--version` prints the product version. A mistyped option name produces a suggestion, and arguments can be read from a response file (`@file`) when a command line becomes too long.

Three short option aliases exist: `-s` for `analyze --solution`, `-o` for `--output`, and `-f` for `--file`. They are not registered uniformly across verbs; in particular, `export --solution` has no `-s` alias.

Results are written to standard output; errors, warnings and notes normally go to standard error. One current exception is the multiple-installation warning from `aicb init`, which is written to standard output. Both streams are set to UTF-8 without a byte order mark at startup, so redirected output stays valid UTF-8. `aicb mcp` reserves standard output for the JSON-RPC channel and writes everything else to standard error.

9.2 Exit codes

The same convention applies to every verb, and it is the interface a script should key on:

Code	Meaning
`0`	Success.
`1`	User error: invalid arguments, missing files, unknown ids.

Code	Meaning
`2`	Runtime error: an unexpected exception, or a partial import.
`3`	Cancelled (Ctrl+C).
`4`	`migration_failed`: the schema of the session or configuration database could not be migrated.
`5`	`msbuild_not_found`: the MSBuild bootstrap failed.
`6`	`quality_gate_failed`: `analyze --fail-on` matched. The Markdown is still written.

Verb	Codes you can receive
`init`	`0`, `1`
`analyze`	`0` - `6`
`export`	`0`, `1`, `2`, `3`, `4`
`import`	`0`, `1`, `2`, `3`, `4`
`list`	`0`, `1`, `2`, `4`
`mcp`	`0`, `1`, `2`
`call`	`0`, `1`, `2`, `3`

`export`, `import` and `list` never return `5`, because they need no MSBuild. Only `analyze --fail-on` returns `6`, and `init` returns only `0` or `1`.

9.3 `aicb init`

Wires `aicb` into a project directory so that an MCP client can find it. It writes three kinds of artifact: the `.mcp.json` server entry, the agent skills under `.claude/skills/`, and the symbol guard together with the harness configuration that loads it. No database, no Roslyn and no MSBuild are involved - this is the step before any of that.

Option	Values	Default	Meaning
`--path <dir>`	directory	current directory	The project directory to wire up.
`--force`	flag	off	Overwrite artifacts that are already there. Without it, an existing `aicb` entry or skill file is left untouched, which is what makes re-running safe.
`--skills <context|all>`	`context` or `all`	`context`	Which agent skills to write. `context` writes the aicb-csharp-context skill only; `all` adds the aicb-code-review and aicb-code-simplifier review pair plus aicb-usage-check.
`--hooks <auto|none|all|claude-code|codex|opencode>`	one of the listed values	`auto`	Which agent harnesses get the symbol guard. `auto` installs it for the harnesses already used in this project.

Example:

```
cd C:\repo\MyApp
aicb init --skills=all --hooks claude-code
```

Behavior:

- `--hooks` is validated before anything is written. An unknown value is exit `1` with the accepted values listed.
- `--path` must exist; otherwise exit `1`.
- Each artifact is reported as one line: `created`, `updated`, `kept` or `refused`, followed by the path and a short detail. If at least one artifact is refused, `init` reports an error and returns `1`. A refusal always means an existing file that `init` will not risk rewriting - an unusable `.mcp.json`, a harness wiring it cannot read, or a path it may not write - so the fix is yours: repair the file, or point `--path` elsewhere.
- When `--hooks auto` finds no harness marker in the project, no guard is installed, the server is registered in `.mcp.json` only, and the command says so explicitly, together with the way to wire a harness anyway (`--hooks claude-code|codex|opencode`, or `--hooks all`).
- Once a guard is installed it *blocks*: a C# symbol search is refused and redirected to the aicb tools. The output names, per harness, the wiring file whose `aicb` entry removes the guard again. `--hooks none` writes no enforcement on the next run; it does **not** remove an installation that is already there.
- Skills are written to `.claude/skills/`. For Codex and OpenCode - detected, or named with `--hooks` - they are also written to `.agents/skills/`, and the server is registered in `.codex/config.toml` or `opencode.json`, because neither reads the shared `.mcp.json`. `--hooks none` skips only the guard; this registration still happens.
- After writing `.codex/config.toml` the output says that Codex reads it only in a project you have trusted in Codex.
- The Codex entry carries `tool_timeout_sec = 300`: Codex gives up on a tool call after 60 seconds by default, and the first call on a large solution analyzes it, which can take longer. An `aicb` entry that is already there is left as it is; `aicb init --force` rewrites it with the line, or add the line by hand.
- If more than one `aicb` installation is found, a warning lists them: every MCP client starts the *first* one on the search path, so updating another one changes nothing a client runs. The warning names how to remove the extra installation.
- The last line is always the same reminder: `Restart or reconnect your MCP client, then call server_info to confirm it took.`

Note: `init` has no preview mode. `--force` is broader than "overwrite what is already there": it also replaces an `aicb` entry that you deliberately pointed at a wrapper script or a debug build, and it rewrites the harness wiring back to its canonical form.

Note: the harness detection on `auto` is deliberately conservative. A `CLAUDE.md` in the project directory counts as evidence of Claude Code. Without one, a Claude Code user whose `.claude/` folder contains only their own `skills/` directory is not detected, because the detection must not read a marker that `init` itself created. The message `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).` is not a failure - pass `--hooks claude-code` explicitly.

The concepts behind these artifacts - what the symbol guard does, how the MCP client is wired, and how to remove it - are described in the MCP manual.

9.4 `aicb analyze`

Runs the Roslyn analysis on a solution and writes the context Markdown. Optionally it reuses a stored snapshot instead of analyzing again, writes a snapshot itself, renders through a run template, and enforces a quality gate.

Option	Alias	Required	Default	Meaning
`--solution <path>`	`-s`	yes	-	The `.sln` file to analyze.
`--output <path>`	`-o`	yes	-	Exact path of the generated `.md`. Missing parent directories are created.
`--session-db <path>`		no	-	An existing SQLite session database. When its stored fingerprint matches the solution, the stored analysis is reused and the Roslyn run is skipped.
`--emit-session-db <path>`		no	-	Path at which the session database is written after the run. Created if it does not exist.
`--force`		no	off	Bypass the snapshot mode and always run a full Roslyn analysis.
`--layer-profile <path>`		no	role-based heuristic	Path to a layer-mapping profile (JSON).
`--run-template <id>`		no	full default render	Id of a run template (built-in or your own).
`--format <tag|yaml>`		no	see below	Inner notation of the `.md` file. The file name stays `.md`.
`--fail-on <expression>`		no	no gate	Quality gate; exit `6` when the expression matches.

Example:

```
aicb analyze -s C:/repo/App.sln -o C:/out/app-context.md \
  --emit-session-db C:/out/app.acb \
  --run-template manual-refactoring \
  --format yaml \
  --fail-on "critical>0 OR debt>120min"
```

What a run prints

On success, standard output carries a small report. Without a stored profile:

```
No layer profile - using role-based heuristic.
Solution analyzed: MyApp
Projects: 12
Wrote context MD: C:\out\app-context.md
Session DB updated: C:\out\app.acb
```

With `--layer-profile`, the first line names the profile that was loaded. With a snapshot hit, the first line becomes `Snapshot hit (file-set hash v2:A1B2C3D4E5F60718); skipping Roslyn analysis.`, and if the emitted database is the one that was read, the last line becomes `Session DB already current; no new snapshot written.` A passing quality gate prints `Quality gate passed (--fail-on "<expression>").`

Validation, in this order

Path, format and other inputs listed below are checked before the expensive work starts. The `--run-template` id is an exception: it is resolved after analysis, so an unknown id can fail only after the full run:

1. `--solution` is missing or does not exist → `error: solution not found: <path>`, exit `1`.
2. `--output` is missing → exit `1`.
3. `--session-db` or `--emit-session-db` is a directory → exit `1`.
4. `--session-db` is given, `--emit-session-db` is not, and the file does not exist → exit `1`, with a message naming both ways out. Reading from a cache that is not there is treated as a typo rather than silently creating the file.
5. `--output` is a directory → exit `1`.
6. The `--layer-profile` file does not exist → exit `1`.
7. `--format` is neither `tag` nor `yaml` (trimmed, case-insensitive) → exit `1`. This is deliberately strict: an unrecognized value would otherwise render the tag notation silently *and* override the format configured on the run template.
8. `--fail-on` was given without an expression (the shape a CI script produces from an unset variable) → exit `1`. An invalid expression is parsed and rejected before the analysis starts.

The run itself

1. **MSBuild bootstrap.** If MSBuild cannot be registered, the run ends with exit `5`.
2. **Snapshot decision**, only when `--session-db` points to an existing file. The file-set fingerprint of the solution is compared against the fingerprint stored with the latest snapshot. The fingerprint covers the solution's `.cs`, `.csproj`, `.xaml`, `.cshtml` and `.razor` files, ignoring `bin`, `obj`, `.vs`, `node_modules`, `packages` and `TestResults`. A snapshot is reused only when it carries an analysis, was produced by a compatible version of the analysis engine (payload schema and analyzer identity), and the source files have not changed. Because the fingerprint uses file timestamps, a difference is first checked against a content fingerprint - so a branch switch or a save without a content change does not force a re-analysis. On a hit the run prints `Snapshot hit (file-set hash <hash>); skipping Roslyn analysis.` and skips Roslyn. On a miss it says why, for example: - `Force flag set; running full Roslyn analysis.` - `No usable snapshot in session DB; running full Roslyn analysis.` - `Snapshot payload-schema mismatch; running full Roslyn analysis.` - `Snapshot analyzer-identity mismatch; running full Roslyn analysis.` - `File-set hash drift (current <a>, snapshot <b>); running full Roslyn analysis.`
3. **Analysis.** Without a snapshot hit, Roslyn analyzes the solution. The layer profile precedence is: explicit `--layer-profile` > the default layer profile configured in the database > the `.aicb.json` file next to the solution > the role-based heuristic. Namespace exclusions from the session database or from the `.aicb.json` file are applied and announced.
4. **Render and write.** The document is written to the exact `--output` path. The output-format precedence is: explicit `--format` > the format configured on the run template > `yaml` on the template-less path. `yaml` is the product default; both notations carry the same content, and the file name stays `.md`.
5. **Emit.** With `--emit-session-db`, the analysis is stored as a snapshot carrying the file-set fingerprint, the payload-schema stamp, the code-metric roll-up and the estimated technical debt with its rating. If the emitted database is the one that was read and the run reused its snapshot, nothing is written.
6. **Quality gate**, only with `--fail-on`, and after the Markdown has been written - a failing gate still leaves you the document.

What the output tells you about itself

The verb reports on standard error every case in which a flag has no effect or in which a configuration source other than the command line drives the run:

- `--force` without `--session-db` → `note: --force has no effect without --session-db (there is no snapshot to bypass).`
- `--layer-profile` on a snapshot hit → `note: --layer-profile is ignored on a snapshot hit; pass --force to re-analyze with it.`
- The analysis scope from the `.aicb.json` file (`analyzePreferredTfmOnly`) on a snapshot hit: the scope is not part of the file-set fingerprint, so the snapshot may have been analyzed under a different scope. Pass `--force` to re-analyze with the current one.
- When the `.aicb.json` names `designTimeBuildPlatform`: before the analysis, the platform it sets (a refused value as a warning, one not checked against an unreadable solution file as such); on a snapshot hit, that the key is not part of the file-set fingerprint.
- Namespace exclusion patterns applied from the session database, or from the `.aicb.json` file next to the solution.
- Analysis limited to the preferred target framework per project.
- A default layer profile configured in the database drives the rendered document and the gate.
- A test profile deviating from the default drives test detection.
- Quality producers that failed and were skipped: `gate counts may be incomplete`.

The quality gate `--fail-on`

The expression grammar is:

```
expression   := and-expression ( "OR" and-expression )*
and-expression := comparison ( "AND" comparison )*
comparison   := metric operator number [unit]
operator     := > | >= | < | <= | = | == | !=
```

Metric names and operators are case-insensitive. `AND` binds stronger than `OR`, and there are **no parentheses** - an expression such as `(critical>0 OR warning>10) AND debt>60min` is rejected as a user error (exit `1`). The number is an integer; an optional unit suffix such as `min` is accepted and ignored, so `debt>120min` means 120 minutes.

Metric	Meaning
`critical`, `warning`, `info`, `ok`	The insight counts per severity.
`debt`	The estimated total remediation effort in minutes.
`complexity-max`	The highest cyclomatic complexity of any method.
`ce-max`	The highest efferent coupling of any type (its outgoing dependencies).
`ca-max`	The highest afferent coupling of any type (its incoming references).
`methods`	The number of methods.
`types`	The number of types.

Both sides of an `AND` or `OR` are always evaluated, so a failing combination can name every clause that matched. Only the clauses that actually contributed are reported:

```
quality gate failed: debt>500 (--fail-on "critical>0 AND warning>10 OR debt>500").
```

The profiles the gate uses are resolved, not hard-coded: in the CLI/headless graph the active MCP profile's `TemplateId` is considered first, followed by that context template's quality-profile slot, the application setting, and the built-in default. The layer profile follows the precedence above, and the test profile is the one active in the database. Without a database, or on any failure to read one, the gate safely falls back to the built-in defaults and stays functional.

9.5 `aicb export`

Re-renders the context document from a persisted session database, **without** running Roslyn. No MSBuild bootstrap is involved, so exit code `5` is unreachable here.

Option	Alias	Required	Meaning
`--session-db <path>`		yes	The database to re-render from.
`--output <path>`	`-o`	yes	Exact output path. Missing parent directories are created.
`--run-template <id>`		no	The same template-driven render as `analyze --run-template`.
`--solution <name|path>`		no	Selects one solution in a database that holds several.
`--format <tag|yaml>`		no	Same precedence as `analyze`.

Example:

```
aicb export --session-db C:/out/app.acb -o C:/out/app-context.md --solution App
```

A session database - in particular the shared default configuration database that the desktop application fills for every loaded solution - can carry more than one analyzed solution:

- Exactly one analyzed solution → it is rendered.
- None → user error (exit `1`) with a pointer to create one first (`aicb analyze ... --emit-session-db <db>`).
- More than one **without** `--solution` → user error with a listing, never a silent guess.
- `--solution` prefers exact matches: the full canonical path, the solution name (file name without `.sln`), or the file name, all case-insensitive. Only when there is no exact match does it fall back to a case-insensitive substring match on the path - so `App` matches `App` and not also `AppCore`. A selector that matches several entries is an error with the matches listed.

On success it prints:

```
Re-rendered from session DB (no Roslyn): C:\repo\App.sln
Projects: 12
Wrote context MD: C:\out\app-context.md
```

9.6 `aicb import`

Imports a constellation JSON - a bundle of templates, presets, profiles and app-settings keys - into a target SQLite database. Three phases: read, preview, apply. No Roslyn and no MSBuild.

Option	Alias	Required	Default	Meaning
`--file <path>`	`-f`	yes	-	The constellation JSON file to import.
`--db-path <path>`		yes	-	The target database: the master-entity/configuration database, explicitly **not** a session snapshot database. Created if it does not exist.
`--mode <SkipExisting|Replace>`		no	`SkipExisting`	Behavior on id conflicts. `SkipExisting` leaves existing entries untouched; `Replace` overwrites them. Parsed case-insensitively; an empty value counts as `SkipExisting`.
`--preview`		no	off	Dry run: show the preview and write nothing.

Example:

```
aicb import -f C:/share/team-constellation.json --db-path C:/out/app.acb --mode Replace --preview
```

Behavior:

- The per-section preview is printed **always**, with or without `--preview`: `Preview for 'Team defaults': RunTemplate: new=2, conflicts=1, unchanged=3, skipped-built-in=4 app-settings keys: 2`
- `--preview` stops after the preview and writes nothing.
- If the file carries app-settings keys, a note on standard error points out that their JSON-backed values can also touch the global app-settings file.
- A file whose preview finds a problem - a tag schema field whose source path is not allowed, an item without an id, an app-settings key that is not allowed or does not convert - is refused before anything is written: the error ends with `The preview found N problem(s) in the file, so nothing was imported. Fix them and import the file again.`
- Past the preview the import is **not transactional**. An item that still fails when it is saved is named in a warning, the warnings end by saying how many items were written and stay written, and the exit code becomes `2` so that a script can detect a partial import. A fully successful import prints `Imported into <path>: applied=<n>, skipped=<n>, app-settings-keys=<n>.`

9.7 `aicb list`

Lists built-in and custom master data. This verb is read-only; there is no verb that creates, changes or deletes master data.

Parameter	Required	Meaning
`kind` (positional)	yes	The master-entity type to list - see the table below.
`--json`	no	JSON instead of the table.

Parameter	Required	Meaning
`--db-path <path>`	no	Source database. Without it, only the built-ins are listed.

The fourteen kinds (compared case-insensitively):

Kind	Content
`run-templates`	Run templates.
`detail-presets`	Detail presets.
`model-profiles`	Model profiles.
`prompts`	Prompts.
`md-profiles`	Markdown profiles.
`tag-schemata`	Tag schemata.
`test-descriptions`	Test descriptions.
`context-templates`	Context templates.
`expansion-strategies`	Expansion strategies.
`pipeline-profiles`	Pipeline profiles.
`quality-profiles`	Quality profiles.
`mcp-profiles`	MCP profiles.
`test-profiles`	Test profiles.
`compression-rules`	Compression rules.

Examples:

```
aicb list run-templates --db-path C:/out/app.acb
aicb list mcp-profiles --json
```

Behavior:

- Without `--db-path`, a throwaway database is created in a temporary directory, migrated and deleted afterwards, so only the built-in entries appear. The table header then says `(built-ins only - pass --db-path to include custom)`.
- `--db-path` must point to an existing file. Unlike `import --db-path`, `list` does not create it.
- The table is one line per entry: two spaces, the id, `-`, the name, and the flags in brackets - `built-in` or `custom`, extended by `, overridden` and `, hidden` where they apply. Rows are sorted by id (case-insensitive) so the output is stable and column-aligned. A closing line states the counts, for example `14 entries (9 built-in, 5 custom).`
- `--json` prints a JSON array of objects with the keys `Id`, `Name`, `IsBuiltIn`, `IsOverridden` and `IsHidden`.
- An unknown kind is exit `1` and prints the full list of valid kinds.

9.8 `aicb mcp`

Starts the MCP server: Model Context Protocol over stdio JSON-RPC, for Claude Code, Cursor, Cline and other MCP clients. The server concepts - profiles, tool pools, sessions and the agent artifacts - are described in the MCP manual; this section covers the command's options and runtime behavior.

Option	Required	Default	Meaning
`--db-path <path>`	no	the standard configuration database is resolved automatically	The master/configuration database. With it the server is profile-aware: rendering follows the active MCP profile, the exposed tool set follows its selection, and its skill shapes the server instructions.
`--mcp-profile <id>`	no	the profile stored as active in the configuration database	Pins the active MCP profile for this server process only.

Example:

```
aicb mcp --db-path "%APPDATA%\AIContextBuilder\user-data\aicb.acb" --mcp-profile mcp-profile/full
```

Behavior:

- **Standard output is reserved for the protocol.** Everything this verb prints goes to standard error.
- Without `--db-path`, the standard configuration database - the same one the desktop application uses, `%APPDATA%\AIContextBuilder\user-data\aicb.acb` - is resolved and, on first start, created and migrated. Which database is in effect is always announced on standard error, never silently. If that resolution fails, the server warns and starts without a profile.
- An explicitly given `--db-path` that does not exist is a **warning**, not a silent creation: a typo must not produce a new database. An automatically resolved standard database *is* created on first start.
- A failed schema migration is a warning, and the server starts without a profile.
- `--mcp-profile` is the one rule of this verb that does not fail open. An unknown or hidden profile id, a missing configuration database or file, or a failed migration makes the server **refuse to start** with exit `1`, listing the valid ids on standard error. Serving a different profile than the operator asked for - silently, over stdio, where nobody reads a warning - is the failure this prevents. The pin is process-local and writes nothing back: a second server or the desktop application keeps its own active profile.
- On a successful profile resolution, the server reports on standard error which profile it serves, whether it was pinned, the effective tool set and whether a skill is attached.
- A failure to register MSBuild is not fatal: a warning is printed, `analyze_solution` and `refresh_session` will fail, and the remaining tools stay usable.
- The server watches two kinds of drift: a configuration drift (a profile edited in the desktop application while this server runs is detected and reported on `server_info`; restart or reconnect the server to load it) and a version drift (this binary compared against the database it read, reported on `server_info`).
- Every MCP tool call is recorded in the `tool_calls` table of the configuration database, fail-open. A server started without a database records nothing.
- The server has **no** `--solution` option: the solution comes with each tool call, and a `.sln` path passed to a session-taking tool initializes the session on first use.
- One binary serves both current MCP protocol revisions over stdio: `2026-07-28` via `server/discover` (stateless) and `2025-11-25` via the `initialize` handshake. Only stdio is supported - there is no HTTP or SSE transport, so session headers, resumability and OAuth of the newer revision do not apply here.

9.9 `aicb call`

Invokes **one** MCP tool exactly once and prints its result. No server, no client, no JSON-RPC handshake. This is useful for validating a tool live after an update, and as a plain shell one-liner.

Parameter	Required	Meaning
`tool` (positional)	yes	The MCP tool name, for example `find_usages`, `server_info` or `batch`.
`--sln <path>`	no	An absolute `.sln` path. It is bound to the tool's `sessionId` / `solutionPath` argument and analyzes the solution once (self-init). Omit it for a tool that needs no session, such as `server_info`.
`--arg <name=value>`	no, repeatable	One tool argument per occurrence.
`--db-path <path>`	no	An optional configuration database. Without it the tool runs database-free.

Examples:

```
aicb call server_info
aicb call find_usages --sln C:/repo/App.sln --arg symbol=OrderService
aicb call find_by_side_effects --sln C:/repo/App.sln --arg effect=io --arg includeTests=true
aicb call batch --sln C:/repo/App.sln --arg queries='[{"tool":"find_usages","args":
{"symbol":"Foo"}}]'
```

Behavior:

- Each value needs its own `--arg`; a single `--arg` token carries exactly one `name=value` pair. A token without `=`, or with an empty name, is a user error (exit `1`). Repeated names: the last one wins.
- Values are converted to the parameter type: `bool`, `int`, `long`, `double`, enums (case-insensitive) and string arrays. A string array is written as a JSON array or as a comma-separated list. Any other, complex argument is passed as JSON.
- Numbers are read with the invariant culture: `.` is the decimal separator, not `,`. A value such as `0,6` is rejected with a message that names the cause and the fix.
- Dispatch is generic over the registered MCP tools, so a tool that was just added is callable as soon as it ships, and **every** tool is reachable - including tools outside the default profile's pool and tools that write, such as `install_agent_hooks`. This is the one place where the profile's tool selection does not restrict you; use it deliberately.
- `--db-path` applies the same configuration as `aicb mcp --db-path`, so database-defaulting tools and the self-init pick up the per-solution configuration of the desktop application.
- Standard output carries the tool result, so it can be piped or redirected; diagnostics go to standard error.
- An unknown tool or a missing or invalid argument is reported as the tool's own message, exit `1`. Any other unexpected failure is exit `2`.
- A failure to register MSBuild is only a warning; tools that need no solution still run.

9.10 Environment variables

The following variables are read by the shipped product. Set them for the process that starts the server; they do not persist anywhere.

```
# Windows (PowerShell)
$env:AICB_MCP_TOOLS = "all"

# Linux / macOS
export AICB_MCP_TOOLS=all
```

Variable	Values	Default	Effect
`AICB_MCP_TOOLS`	`all`, `lean`, empty, or a selection list	the active MCP profile's tool pool	Assembly-wide tool curation: which tool groups the server exposes. `all` also exposes the opt-in infrastructure tools (session, memory, DB-entity and API-compatibility tools) that no profile menu lists. Literal `lean` selects the complete 72-tool lean core. Empty or unset means no override, so the active profile decides (54 tools in the default profile). A selection list is either a comma-separated list of tool-group names or the `methods:<tool>,<tool>` form naming individual tools. Unknown group names are ignored and a group list with no matches falls back to the lean core; a `methods:` list is used literally, so unknown names can produce an empty tool set and do not trigger that fallback. An explicit spec is static: it does not change when a profile is switched.
`AICB_MCP_SESSION`	comma-separated `key=value`, keys `ttlMinutes`, `maxSessions`, `sweepMinutes`	TTL 90 minutes, at most 8 sessions, sweep every 5 minutes	Fine-tuning of the in-memory session cache. The TTL is sliding: every tool access resets the clock. The session limit is the hard memory bound; on overflow the least recently used session is evicted. Invalid or non-positive values are ignored per key, and unset keys keep their default.

Variable	Values	Default	Effect
`AICB_MCP_AUTO_REFRESH`	`Off` / `0` / `false` / `no`, `Reactive`, `Proactive` / `1` / `on` / `true` / `yes`	the active profile's value; profiles the product creates use `Reactive`	Whether a stale MCP session is re-analyzed before a read tool answers, and whether file watchers are armed. `Reactive` repairs a drifted session when a question arrives; `Proactive` additionally watches the solution folder in the background. Missing, empty or unrecognized values mean no override. See "Sessions and staleness" in the MCP manual for the modes.
`AICB_MCP_DEBOUNCE_MS`	positive integer (milliseconds)	15000 (15 seconds)	Debounce of the file watcher in `Proactive` mode: how long the watcher waits for quiet before it refreshes.
`AICB_MCP_STALENESS_WINDOW_MS`	positive integer (milliseconds)	5000 (5 seconds)	Throttle window of the staleness check, so a burst of tool calls does not repeat the directory walk for each one.
`AICB_MCP_LOAD_WAIT_MS`	positive integer (milliseconds)	45000 (45 seconds)	How long a tool call waits at the solution registry's load gate before it is told what is holding it up, instead of blocking. Only the MCP server applies this bound; the desktop application and the other CLI verbs wait without one.
`APPDATA`	a directory path	the platform's application-data folder	The root of `%APPDATA%\AIContextBuilder`, from which every settings and database path is derived. An explicitly set value wins on all platforms. On Linux and macOS the same expansion resolves `%VAR%` tokens and converts `\` to `/`.
`DOTNET_gcServer`	`0` disables the server garbage collector	server GC is on	The product's builds enable the server garbage collector, and the packed tool carries that setting with it. Setting this variable to `0` turns it off for one process - the environment variable beats the build setting. Use it on a memory-constrained machine.

`aicb.mcp.json`

`aicb.mcp.json` is the file-based twin of the two MCP environment variables. It lives in the user configuration directory - `<ApplicationData>/AIContextBuilder/aicb.mcp.json`, on Windows `%APPDATA%\AIContextBuilder\aicb.mcp.json` - and is intended for headless operation without the desktop application.

It is **hand-written and only read**; the server never writes it. Do not confuse it with the `.mcp.json` in your project directory, which `aicb init` writes.

All fields are optional; a field that is not set means "no override". Precedence is: built-in defaults < `aicb.mcp.json` < environment variable. The environment variables remain the quick operator override.

```
{
  "toolSpec": "all",
  "sessionCache": { "ttlMinutes": 120, "maxSessions": 4 }
}
```

- `toolSpec` uses the same grammar as `AICB_MCP_TOOLS`.
- `sessionCache` accepts `ttlMinutes`, `maxSessions` and `sweepMinutes`; only set, positive values are applied.
- The file is read fail-open: if it is missing, unreadable or malformed, the server starts on the built-in defaults and logs a warning. That warning is the only thing that distinguishes a broken file from a missing one, so check the server's standard error if a setting appears to have no effect.
- Parsing is case-insensitive and tolerates comments and trailing commas, because the file is written by hand.

9.11 Typical headless workflows

Analyze a solution

```
aicb analyze --solution C:/repo/App.sln --output C:/out/app-context.md
```

`--solution` and `--output` are required. All inputs are validated before the analysis starts, so a mistyped path is reported immediately as a user error (exit `1`) rather than after a full run.

Analyze once, reuse the analysis

Store the analysis in a session database and let later runs reuse it while the source files are unchanged:

```
# First run: analyze and store the result
aicb analyze -s C:/repo/App.sln -o C:/out/app-context.md --emit-session-db C:/out/app.acb

# Later runs: reuse the stored analysis when the sources are unchanged
aicb analyze -s C:/repo/App.sln -o C:/out/app-context.md --session-db C:/out/app.acb
```

A run that hits the snapshot prints `Snapshot hit (file-set hash <hash>); skipping Roslyn analysis.` and skips the expensive step. Pass `--force` when you want to re-analyze even though the snapshot matches - for example after passing a different `--layer-profile`, which is ignored on a snapshot hit.

Re-render without a new analysis

When only the notation or the run template changes, re-render from the stored database:

```
aicb export --session-db C:/out/app.acb -o C:/out/app-context.md --format tag
aicb export --session-db C:/out/app.acb -o C:/out/app-context.md --run-template manual-refactoring
```

`export` runs no Roslyn and needs no MSBuild. It renders the most recent snapshot of each analyzed solution in the database.

The shared database of the desktop application can contain several analyzed solutions. Name the one you want:

```
aicb export --session-db "%APPDATA%\AIContextBuilder\user-data\aicb.acb" --solution App -o app.md
```

Without `--solution`, a database with more than one analyzed solution is a user error with a listing - the tool never guesses.

A quality gate in CI

```
aicb analyze \
  --solution App.sln \
  --output artifacts/context.md \
  --emit-session-db artifacts/context.acb \
  --fail-on "critical>0 OR debt>120min"
```

The Markdown is written before the gate is evaluated, so a failing gate still leaves you the document for the build artifacts. Only the exit code changes to `6`.

A shell script can distinguish the outcomes precisely:

```
aicb analyze -s App.sln -o context.md --fail-on "critical>0 OR debt>120min"
status=$?
case $status in
  0) echo "analysis complete, gate passed" ;;
  6) echo "gate violated - context.md was still written" ;;
  1) echo "bad invocation - fix the arguments" ;;
  5) echo "MSBuild not found on this machine" ;;
  *) echo "aicb failed with exit code $status" ;;
esac
```

The same gate as a GitHub Actions step:

```
- name: Analyze and enforce the quality gate
  run: |
    aicb analyze \
      --solution MyApp.sln \
      --output artifacts/context.md \
      --emit-session-db artifacts/context.acb \
      --fail-on "critical>0 OR debt>120min"

- name: Upload the context document
  if: always()
  uses: actions/upload-artifact@v4
  with:
    name: aicb-context
    path: artifacts/context.md
```

`if: always()` is what makes the second step run even when the gate failed - the document exists in both cases.

The step above assumes that the job has already set up the .NET 10 SDK, installed `aicb` and restored the solution. The composite action at the root of the public repository does those three things and then runs the same command:

```
- name: Analyze and enforce the quality gate
  uses: gregordadera/aicb-roslyn-mcp@main
  with:
    solution: MyApp.sln
    output: artifacts/context.md
    fail-on: "critical>0 OR debt>120min"
    args: --emit-session-db artifacts/context.acb
```

Input	Default	Meaning
`solution`	required	The `.sln` file, passed as `--solution`.
`output`	`aicb-context.md`	Passed as `--output`.
`fail-on`	empty, no gate	Passed as `--fail-on` when not empty.
`version`	empty, latest	The version of the `aicb-roslyn-mcp` package to install.
`dotnet-version`	`10.0.x`	The SDK set up with `actions/setup-dotnet`; empty uses the SDKs already on the runner.
`restore`	`true`	Runs `dotnet restore` on the solution first.
`args`	empty	Further `analyze` options, split on whitespace; a value containing a space cannot be passed this way.

The exit code of `analyze` is passed through unchanged, so a failed gate fails the step with `6`, and the document is written as before - the upload step with `if: always()` works the same way behind the action. The action's outputs are `output` (the path passed in) and `exit-code`, which a workflow that runs the step with `continue-on-error:`

`true` can read to tell a failed gate (`6`) from a broken invocation. The tool is installed into a directory of its own under the runner's temporary folder, not as a global tool, so it does not collide with an `aicb` the workflow installs itself. Pin the action to a commit SHA, or to a release tag that already contains `action.yml`, rather than `@main` when the build depends on it.

Analyze several solutions in a loop

```
for sln in src/*.sln; do
  name=$(basename "$sln" .sln)
  aicb analyze -s "$sln" -o "out/$name.md" --emit-session-db "out/$name.acb" || exit $?
done
```

Each solution gets its own document and its own session database. The exit code is propagated, so the loop stops at the first failure.

Query the code from a shell

`aicb call` answers a single question without starting a server:

```
aicb call find_usages --sln C:/repo/App.sln --arg symbol=OrderService
aicb call get_diagnostics --sln C:/repo/App.sln --arg severityFloor=error
```

The result goes to standard output and can be piped or captured. The command reaches every MCP tool, including ones outside the default tool pool.

What the command line does not do

The CLI is the tool for repeating and automating; the desktop application is the tool for choosing and curating. The following exist only in the desktop application and have no CLI equivalent:

- Creating, editing or deleting master data. `aicb list` is read-only; to bring master data in, create it in the application or import a constellation JSON with `aicb import`.
- LLM runs (sending a document to a model) and their results.
- The interactive selection tree, detail levels and section selection, and the prompt context (role, goal, constraints).
- Switching the active database and managing sessions and snapshots interactively.
- Exporting a constellation. Importing one works from both sides; exporting it is only available in the application.

Both sides meet in the database and in the render path: a solution analyzed in the application can be re-rendered with `aicb export`.

10 Data, storage and privacy

AIContextBuilder keeps everything it knows in a local SQLite database and a small number of files around it. Nothing is transmitted anywhere unless you send it yourself. This chapter describes what is stored, where it lives, how to move it to another machine, how to remove it, and what the product does - and deliberately does not do - with your information.

10.1 Sessions and snapshots

What a session stores

A session is a saved working state of the Context Builder for one solution. It holds:

- a name and the solution it belongs to,
- creation and last-modified timestamps,
- an optional pinned snapshot (if none is pinned, the session is not fixed to a particular analysis state),
- the active context template (`default` unless you chose another),
- a by-value copy of all master-data values as they were when the session was created - so reopening an old session does not silently change under a later profile edit,
- the tree selection: the list of nodes whose state is not `Inherit`,
- the last user prompt you entered, so it is reproduced when you reopen the session,
- free-text notes for the session,
- an archive flag, and
- per-session picker overrides from the detail and sections panels (if you never touched them, the normal resolution chain applies).

Node-level overrides are stored separately and belong to the session.

Archiving is **non-destructive**: runs, snapshots and evaluations are kept, and the session simply leaves the default filter. The Sessions tab offers the filters `Active`, `Archived` and `All`. In the session's action area you can `Open in New Tab`, `Export` it as a JSON file (`As JSON file`), `Duplicate` it (`Copy with snapshot`), or `Delete` it (`Permanent - confirms`). Deleting a session is permanent and asks for confirmation first. For the session workflow itself, see "Workspace, sessions and snapshots" in the desktop app manual.

Missing nodes when you open a session

If part of the stored selection or of the per-node detail overrides no longer exists in the current tree - typical after a refactoring - the application shows a window titled `Session opened - missing nodes`. Its summary says how many nodes the session references that no longer exist and how many nodes of the selection were restored, and it lists the missing node keys; hovering an entry shows the full node key. The window has a single `OK` button and is a report, not a question: orphaned nodes are removed from the session on the next save.

What a snapshot stores

A snapshot is a complete analyzed solution, stored as JSON together with metadata: its name, its type, the creation time, a hash of the analyzed file set, the format stamp of the stored payload, and quality figures (technical-debt total and rating, code metrics). Snapshots are what make a comparison against an earlier state possible without re-analyzing the solution.

- **Automatic snapshots.** One is written on every fresh analysis. The setting `Max snapshots per solution` (Settings → General, default `10`) limits how many automatic snapshots are kept; when the limit is exceeded, the oldest ones are deleted. Snapshots you saved yourself are never deleted and do not count toward the limit.
- **Manual snapshots.** In the Snapshots tab, `Create Manual Snapshot` saves one with the name entered in the dialog for the solution currently opened in the Context Builder. Snapshots do not have a notes field. The icon buttons next to it rename and permanently delete the selected snapshot.
- **Comparison.** `Diff Two Snapshots` opens the diff window when exactly two rows are selected - Ctrl-click a second row to enable it. The window shows added and removed types, and per changed type the added and removed methods plus signature changes (return type, visibility, `static` flag, parameter list).

The MCP tools `save_session` and `compare_with_previous` do the same from an agent. `save_session` persists the current session as a named manual snapshot and returns its id. `compare_with_previous` diffs the current session against a previously saved snapshot of the same solution - the newest by default, or the N-th previous via `snapshotsBack` (`0` = newest saved, `1` = the one before it, and so on) - optionally narrowed to a single type and/or method.

What a comparison does **not** do: it is name-based and syntactic. It reports types and method signatures; it does not diff method bodies, so it cannot tell you which lines inside a method changed.

Note: a snapshot keeps the line spans of the analysis it was taken from, but an MCP session recalled from it reports `lineNumbersAvailable: false`: its lines describe the code as analyzed, not the current working tree. This is also stated in the description of the `save_session` tool.

A snapshot records the format it was written in. If a snapshot cannot be read back - an older or damaged payload - it is treated as unusable and the solution is analyzed again instead of returning a half-read result.

Note: a snapshot written by an earlier version is not reused as current. An analysis that would reuse it analyzes again; an MCP session recalled from it is marked `stale`. Where such a snapshot lacks a fact the current version records, the answers that depend on it say so instead of answering as if nothing were missing: `VIEWS NOT KNOWN` (which `.cshtml` views the analysis read), `GENERATED CODE NOT KNOWN` (code a generator did not write, generated C# under `obj/`, markup-only Razor components), `TOP-LEVEL CREATIONS NOT KNOWN` and `TOP-LEVEL EFFECTS NOT KNOWN` (top-level statements). `refresh_remembered` re-analyzes and clears them. A snapshot of the current version carries these facts, so a recalled session answers like a live one.

10.2 The database file

Name, location and path rules

The application's own database ends in `.acb` - not `.db`, not `.sqlite`. Both the start-up path and the `Switch Database...` / `Create New DB...` actions refuse any other extension, and the file picker offers only `AICB database (*.acb)|*.acb`, deliberately without an "All files" entry, so an extension the application rejects cannot be selected in the first place.

The default location is:

```
%APPDATA%\AIContextBuilder\user-data\aicb.acb
```

The Storage page's `Paths` section exposes the base path and shows the app-settings file path. The database path is not an editable UI field; change `Storage.DatabasePath` by hand in `%APPDATA%\AIContextBuilder\app-settings.json` while the application is stopped:

Setting	Default	Meaning
Base path	`%APPDATA%\AIContextBuilder`	Root folder for application data - database, backups, exports.
Database path (`Storage.DatabasePath` in JSON only)	`{BasePath}\user-data\aicb.acb`	The database file itself.

The database path is resolved as follows:

1. If the database path is empty, `<BasePath>/user-data/aicb.acb` is used.
2. Otherwise, environment variables are expanded, then the `{BasePath}` token is replaced (case-insensitively).
3. An absolute path is used as it is; a relative path is resolved relative to the base path.
4. The result must end in `.acb`. A different extension is refused with an error naming the offending path, rather than silently falling back to the default.

The path is re-read on every database open, so a change takes effect without restarting the application. The base path itself takes effect after an application restart. Environment variables in both settings are expanded on Windows as `%VAR%`; on Linux and macOS every `%VAR%` notation is expanded as well and backslashes are rewritten to the platform separator.

If the application does not start because of a database-path problem: open `%APPDATA%\AIContextBuilder\app-settings.json`, check `Storage.DatabasePath`, rename the file if necessary and correct the path.

What the database contains

One file holds the complete state of the product. It has 35 tables, which fall into these areas:

Area	What it holds for you
Solutions and snapshots	The registered solutions and their saved analyses.
Sessions	Your saved working states and the per-node overrides that belong to them.
Runs	Every executed run, its complete settings snapshot, and every LLM call that belongs to it.
Reasoning	Parsed reasoning steps, findings (`Critical` / `Warning` / `OK` / `Info`), and your evaluation of a run.
Master data and profiles	Your own and the built-in prompts, templates, run templates, Markdown profiles and their slots, tag schemata, detail presets, expansion strategies, model profiles, layer profiles, namespace exclusions, compression rules, pipeline profiles, quality profiles, MCP profiles, test profiles and test descriptions.
Insights	Insight results per solution, triage decisions (suppressions) with their history, and queued follow-ups.
Settings and usage	Application settings, the MCP usage recording (see below), your free-text notes on recorded calls, and imported usage reports.

The stored payload of every LLM interaction contains the model's **complete response text**. It contains no API key; of your prompt, only its first line (the goal) is stored with the run. If you pass the database file on or put it into a backup, those model responses travel with it.

Two facts worth knowing about the file itself:

- It is a SQLite database in WAL journal mode, so while it is open two companion files exist next to it: `aicb.acb-wal` and `aicb.acb-shm`. They belong to the database; a backup takes a consistent snapshot of the database instead of archiving them (see "Backups").
- Runs, sessions, snapshots and evaluations hang off a solution. Removing a solution in the application therefore removes its dependent data too, and the confirmation names what that is: the sessions, the snapshots, and "all related runs, run snapshots, LLM interactions, reasoning entries". This cannot be undone.

Note: the database stores runs of the types `Manual`, `Iteration`, `Preselection` and `Pipeline`. Only **Manual** runs can currently be selected in the interface; `Iteration` and `Preselection` are not released yet, and `Pipeline` is a placeholder - it is accepted by the stored data and the repositories, but nothing executes it.

Schema version and migrations

The database carries a schema version. On start-up the application compares it with the version the running build knows and applies the pending migrations automatically. Details:

- Migrations are **forward-only**; there is no down-migration.
- Each step runs in its own transaction. A step that fails leaves the database at its previous version with the change rolled back, so a corrected retry is safe.
- Several instances that start at the same moment against the same database - the desktop application and the MCP servers of parallel sessions - migrate it once: an instance that comes second skips the steps another one already applied.
- A migration takes effect immediately, not when instances are merged: the first start after an update migrates the shared database **for all running instances at once**. Older builds report a mismatch afterwards; this is expected.
- A database written by a **newer** build than the running one must not be opened by it. The desktop application detects this before migrating and exits with a message naming the database schema version and the version this build knows, rather than writing the older shape back. The standalone MCP server tolerates the state, reports the schema deviation through its `server_info` tool, and keeps working. The stderr breadcrumb belongs to the separate configuration-drift check and appears only when `server_info` runs. `Switch Database...` refuses a file with a higher schema version as well.
- If the schema cannot be updated, the desktop application shows the error and exits; the database is left unchanged.

10.3 Backups

Automatic backup is **on by default for a new installation**; an existing installation keeps the value its settings file stores. Its four settings live in Settings → Storage, section `Auto-Backup`:

Setting	Default	Meaning
`Enable auto-backup on app start`	on (new installations)	Master switch.

Setting	Default	Meaning
`Minimum interval (days)`	`7`	Auto-backup only runs if at least this many days have passed since the last backup.
`Keep at most (count)`	`5`	Maximum number of backup archives retained; the oldest are pruned beyond it.
`Last backup`	-	Timestamp of the last successful backup; written by the service, read-only in the interface.

The automatic backup runs in the background once the main window is open, so it does not delay the start; its result appears in the notice bar only when something went wrong. `Backup Now` runs a backup immediately, regardless of the configured interval - useful before risky operations. It also runs in the background, its button is disabled while it runs, and it can be used while an analysis is running. One backup runs at a time. The interval is clamped to a minimum of one day, so the shortest effective interval is one day.

A backup is a ZIP archive of the **base-path hierarchy**, written to `{BasePath}\backups\` with a name of the form `backup-<yyyyMMdd-HHmmss>.zip`. The `backups\` folder itself is skipped while collecting, so an archive never contains older archives. The configured database is added as a **consistent snapshot**, taken with SQLite's own backup function, even when it lives outside the base path (the usual case for an absolute database path). The snapshot is taken while the application and any MCP server on the same database keep working, and it contains every committed change, so the `-wal` and `-shm` companions are not archived. Other database files under the base path - for example old copies you kept there - are left out and counted in the result message.

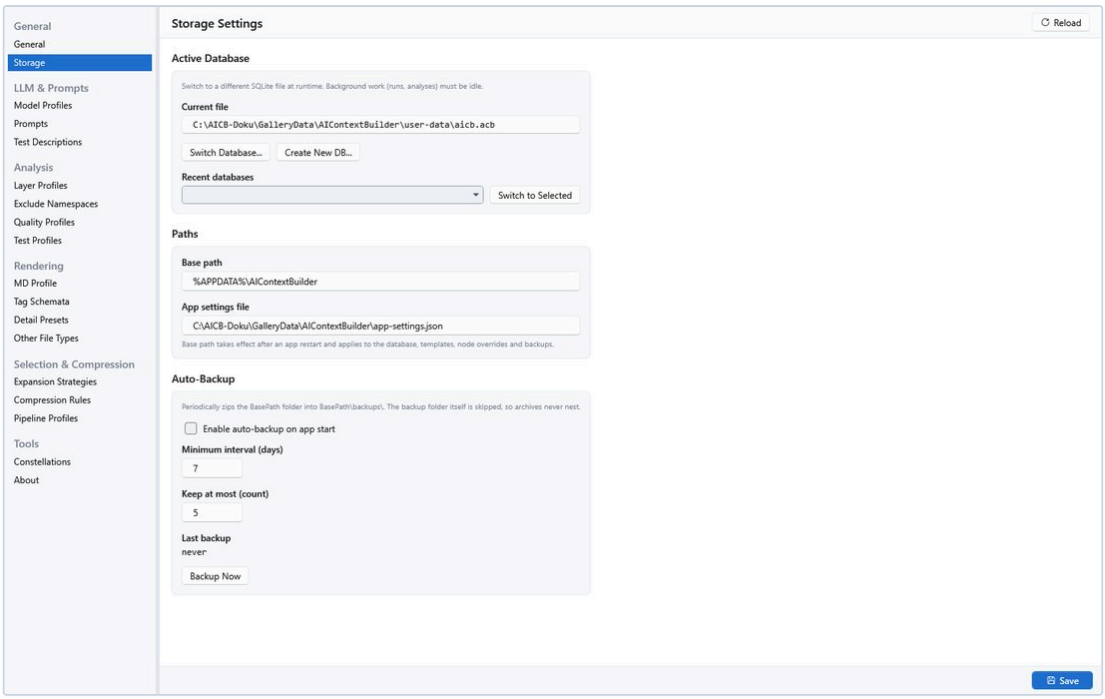
Three properties of the archive matter in practice:

- **A finished archive is named `.zip` only when it is complete.** While it is written it is called `backup-<...>.zip.inprogress`; an interrupted run leaves only such a file, which the next run removes, and never a file that looks like a finished backup.
- **A backup can still be incomplete.** A file under the base path that cannot be read is skipped, and the run reports the status **PARTIAL** instead of success. The archive is still written; its file name carries the suffix `-partial` so the verdict stays visible even after the result message is gone. The retention rule never prunes the newest complete archive. The result message names up to five skipped files and summarizes any further ones as `and N more`.
- **An incomplete run still advances the last-backup timestamp**, so it is not retried at every start.

A backup that runs while you switch the database, create a new one or change the data directory is waited for: those actions are refused with `Cannot switch DB while 1 background task(s) are active.` (the action name varies) until it has finished.

10.4 Switching and creating databases

The `Active Database` section of Settings → Storage lets you work with more than one database file:



Settings > Storage: the active database, paths and auto-backup

- `Current file` shows the full path of the active database; it is read-only there.
- `Switch Database...` picks an existing `.acb` file and makes it active.
- `Create New DB...` creates a fresh, empty database at a location you choose and switches to it - useful for separating projects or starting clean.
- `Recent databases` keeps the recently used paths (the active one excluded); pick one and click `Switch to Selected`.

Rules that apply to both actions:

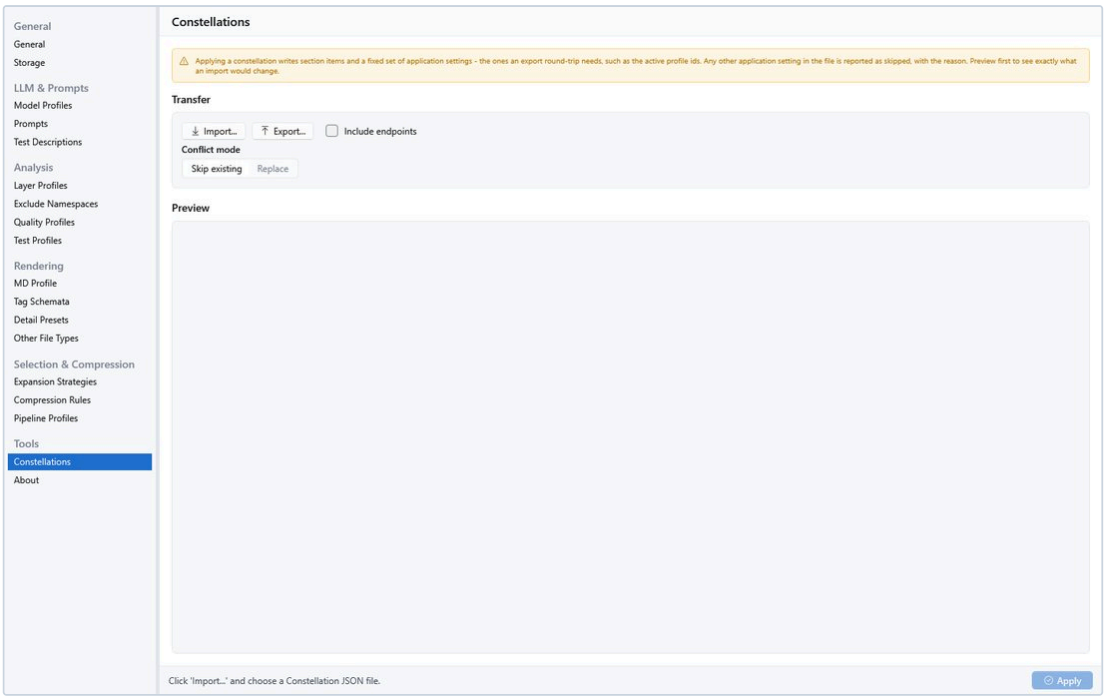
- All background work - runs and analyses - must be idle. A running run blocks the switch.
- The file must carry the `.acb` extension. A wrong extension is rejected before anything is read or written.
- A file whose schema version is **newer** than the running build is rejected.
- `Create New DB...` refuses a path where a file already exists; use `Switch Database...` for an existing database.
- If the target needs migrations, they are applied as part of the switch, after a confirmation. If a migration fails, the previously active database path is restored.
- After a switch, the settings are read freshly from the new file, and the path is remembered in the recent list.

10.5 Import and export

Configuration bundles ("constellations")

A constellation is a JSON file that carries the **portable** half of your configuration - your own master-data entities plus the application settings that make sense on another machine. The use case is "take my configuration to a new PC", or sharing a setup.

You find it in Settings → `Constellations`. The panel offers `Import...`, `Export...`, an `Include endpoints` checkbox, a conflict mode, a read-only `Preview` of what an import would write, and the `Apply` button. A `Last file:` line names the file the last import or export used.



Settings > Constellations: export and import of your configuration as one JSON file

What an export contains. Sixteen groups of master data, in this order: prompts, Markdown profiles, tag schemata, detail presets, test descriptions, model profiles, context templates, expansion strategies, run templates, layer profiles, namespace exclusions, compression rules, pipeline profiles, quality profiles, MCP profiles and test profiles. **Built-in entries are skipped** - only your own entries travel. Model profile endpoints are written only if you tick `Include endpoints`; by default they are redacted so the file is safe to share. **API keys are never exported** - they live in the Windows Credential Manager (see "API keys").

Which application settings travel. Only these keys may be written by a bundle file; any other key in the file is reported as skipped, with the reason:

```
ActiveCompressionRulesProfileId   ActivePipelineProfileId
ActiveQualityProfileId            ActiveMcpProfileId
ActiveTestProfileId               LastSelectedSettingsSubTabLabel
DefaultLayerProfileName           General
ActiveContextTemplateId           DefaultExclusionListName
DefaultExpansionStrategyId        DefaultGraphLayoutEnabled
AutoTriggerInsightsOnLoad         PersistInsightsWithSession
TooltipsEnabled                   TooltipShowDelay
EnableLoadPerfLogging
```

Note: `General` travels as one block. Setting this key **replaces all four of its sections**, so a hand-written bundle that names only one of them resets the other three to their code defaults.

What never travels. These settings are machine-local and are deliberately excluded:

Setting	Why it does not travel
`Storage`	Absolute paths; on the target machine they point at nothing, and the running process has already resolved its database path.

Setting	Why it does not travel
`RecentSolutionPaths`, `RecentDatabasePaths`	Absolute paths.
`Layout`, `ModeProfiles`	Splitter positions in pixels; a layout tuned for one monitor is wrong on the next.
`HasRunFirstAnalyzeOnce`	The first-run marker of *this* installation; an imported `true` would rob a fresh installation of its one automatic insights run.

Version checks on import. Two conditions must hold, or the file is rejected:

1. The schema URL must match exactly: `https://aicontextbuilder.dev/schemas/constellation-v1.json`. A different URL is rejected even if the JSON structure would be compatible.
2. `MinAppVersion` must not be higher than the running application version.

Conflicts. For constellation import, the conflict mode is an always-visible radio group in the panel:

Choice	Effect
`Skip existing`	Existing ids stay untouched; new entries are saved.
`Replace`	Existing ids are overwritten (upsert).

There is no separate constellation conflict dialog or Cancel choice. The dialog with `Skip conflicts`, `Replace conflicts` and `Cancel` belongs to the individual master-data import workflow.

The rule is "the id exists, the mode decides" - not "is the content equal".

No rollback. A file whose preview finds a problem is refused as a whole, with nothing written. Past the preview, an import has neither a rollback per entry nor across the run. Each entity group is written through its own connection, and errors are collected and reported at the end, naming each item that failed. What is guaranteed is the write order: entities first, then the settings that refer to them. A cancelled or failed import can therefore leave a partial state. **Take a backup before importing.**

Older bundle files that still carry the retired keys `FindingActionMode` or `CodeQuality` are read without error: those keys are skipped silently and noted in the log, so old backups remain usable.

The same import is available headless:

```
aicb import --file my-config.json --mode SkipExisting --db-path "C:/data/aicb.acb"
```

`--file` is required. `--mode` is `SkipExisting` (default) or `Replace`. `--preview` is a dry run: it prints the per-section preview (new / conflict / unchanged / built-in) and writes nothing.

Usage reports

The `MCP Usage` page in the main navigation group `MCP` can export and import usage reports as JSON:

- `Export` writes the current aggregate to a file you choose (suggested name `usage-report-<yyyy-MM-dd>.json`). The file carries exactly what the `usage_report` MCP tool returns, so it can be imported on another machine.

- `Import` reads such a file and stores it as a **named snapshot beside the live data** - the name is the file name. Imported reports are never merged into the live numbers; the `Snapshots` region lists them next to the `Current` row.
- `Compare` puts an imported report's tool table beside the live one.
- The delete icon removes an imported snapshot; the source file is not touched.

The intended use is the foreign machine: export a report there, import it here, and look at both sets side by side.

Sessions

A session can be exported as a JSON file from the Sessions tab (`Export` → `As JSON file`). There is no matching import.

What cannot be exported

- **API keys** - by construction, they are not part of any export.
- **Sessions, runs, snapshots and reasoning data** - they belong to a concrete solution on a concrete machine. To move them, take the **database file** with you.
- The machine-local settings listed above.

10.6 Where everything lives on disk

`{BasePath}` is `%APPDATA%\AIContextBuilder` by default. On Linux and macOS, `%APPDATA%` resolves to the platform's application-data folder, with `~/.config` as the last fallback.

Location	What it holds	Survives uninstall	Safe to delete
`{BasePath}\user-data\aicb.acb`	The database: solutions, sessions, snapshots, runs, LLM interactions, reasoning, master data, settings, MCP usage recording.	yes	yes - you lose everything in it
`{BasePath}\user-data\aicb.acb-wal` , `aicb.acb-shm`	SQLite companion files, present while the database is open.	yes	only while nothing has the database open
`{BasePath}\app-settings.json`	Only the storage section (base path, database path, backup settings) plus the most recently used solutions and databases.	yes	yes - it is recreated with defaults on the next start; recent paths and a custom database path are lost

Location	What it holds	Survives uninstall	Safe to delete
`{BasePath}\app-settings.json.<random>.tmp`	Staging file of a settings write. Removed automatically; leftovers from a killed process are swept at the next start.	-	yes
`{BasePath}\app-settings.json.corrupt-<timestamp>`	A preserved copy of a settings file that could not be read.	yes	only once you no longer need it
`{BasePath}\backups\backup-*.zip`	Automatic backups of the base-path hierarchy plus a snapshot of the database; only with auto-backup enabled. A `...-partial.zip` is missing a file that could not be read.	yes	yes, if you do not need the backups
`%APPDATA%\AIContextBuilder\aicb.mcp.json`	Optional MCP server configuration. It is **only read**, never written by the application.	yes	yes - the built-in defaults then apply
`%APPDATA%\AIContextBuilder\aicb.log`	Support log: warnings and errors from the desktop application. Rotating: 1 MiB per file, 3 older files retained (`aicb.log.1` ... `aicb.log.3`).	yes	yes
`%APPDATA%\AIContextBuilder\load-perf.log`	Solution-load phase timings; one line per load, written by the desktop application only.	yes	yes
`%LOCALAPPDATA%\AIContextBuilder\design-time-build-cache\`	Cached design-time build results, one file per solution, so a later load can skip MSBuild (see "How the analysis works"). Written by the desktop app, the CLI and the MCP server; `AICB_DESIGN_TIME_BUILD_CACHE` relocates or disables it.	yes	yes - the next load runs MSBuild and writes it again
Windows Credential Manager, entries `AIContextBuilder.ApiKey:<profile id>`	Your API keys.	yes (operating-system store, per user)	yes, if you want to enter the keys again
`<solution folder>\<Name>.aicb.json`	Per-solution configuration: layer rules, namespace exclusions, test detection, layering policy, suppressed findings, analysis scope.	- (lives with your source code; usually committed)	yes - `Export Config` writes it again from the database

Location	What it holds	Survives uninstall	Safe to delete
Files written by `aicb init` in your project	The `.mcp.json` entry, `.claude/skills/...`, the hook file `.claude/hooks/aicb-symbol-guard.mjs` (or the Codex/OpenCode equivalent) and the guard entry in the harness configuration.	-	they must be **edited**, not simply deleted

Two notes on this table:

- `app-settings.json` always stays at the default location, no matter how `BasePath` is set. If you moved the base path, you have **two** directories to look at.
- The fields for `templates.json` and `node-overrides.json` still exist in `app-settings.json`, but nothing reads or writes them - the corresponding entities live in the database. The effective storage entries are the base path, `Storage.DatabasePath`, and the backup settings. `Storage.AppSettingsPath` is currently inert: the application settings file remains at its default location.

The per-solution configuration file

The file `<SolutionName>.aicb.json` sits next to your `.sln`. It is meant to be committed with your repository. A database-free headless MCP server or CLI run auto-discovers its layer and exclusion fallbacks, its analysis scope and its build platform; other fields have the consumer-specific behavior documented in the configuration chapter. It can contain:

- `layerRules` - the layer mapping (pattern, match type, layer),
- `exclusions` - namespace exclusion patterns,
- `keepNamespaces` and `exclusionsDisabled` - keep rules that are never excluded, and the opt-out that excludes nothing,
- `layeringPolicy` - only written when it is not `Advisory`,
- `testProjectRules` and `testAttributeNames` - the test detection,
- `autoInit` - the opt-in flags that let a fresh clone initialize the corresponding axes automatically,
- `suppressions` - triage decisions (producer, type, member, reason),
- `analyzePreferredTfmOnly` - the analysis-scope opt-in (see "Profiles, master data and solution configuration"; it has no database counterpart and is edited by hand),
- `designTimeBuildPlatform` - the platform of the design-time build, read by the desktop app, the CLI and the MCP server (it has no database counterpart either and is edited by hand).

The file is written atomically through a temporary file of its own, then a replace; a temporary file left by an interrupted write is removed by a later write. It is hand-editable: comments, trailing commas and case-insensitive keys are accepted, and an invalid value fails the whole file rather than silently dropping one axis - with one exception: a malformed entry in `suppressions` costs only that entry. In the Solutions tab, `Export Config` writes the active layer profile, exclusion list, test profile, triage decisions and the analysis-scope value into the sidecar, and keeps a `designTimeBuildPlatform` the file already holds. `Import Config` reads the layer, exclusion and test axes into the desktop database; the remaining fields keep their separate reading paths.

10.7 Removing your data

The product offers exactly **two** delete functions, both in the desktop application:

Function	Where	What it does
`Clear Usage Data`	MCP Usage panel, button `Clear`	Deletes the recorded MCP calls, either all of them or those older than a date you pick. The flow is deliberate: you first pick the scope, then an **export is written before anything is deleted** (cancelling the file picker aborts the whole flow), then a confirmation names the number of calls and warns that notes written on those calls are deleted with them and are not part of the export, and only then are the rows deleted.
`Clear API Key`	Model profiles panel	Removes the stored API key from the Windows Credential Manager. Subsequent runs fail until you enter a new key.

What does **not** exist:

- **No CLI verb deletes anything.** The complete verb list is `init`, `analyze`, `export`, `import`, `list`, `mcp`, `call`. There is no `clean`, `reset`, `uninstall` or `purge`.
- **No MCP tool deletes the usage recording.** `usage_report` is read-only.
- **No global "delete all my data" button.**
- **No automatic undo of `aicb init`.** The command merges its entries into files that are yours; removing them is a manual edit.

A complete manual removal therefore means:

1. Delete the whole `%APPDATA%\AIContextBuilder\` directory. This covers `app-settings.json`, the database and its companions, `backups\*.zip`, `aicb.mcp.json`, `aicb.log` and `load-perf.log` in one step. Delete `%LOCALAPPDATA%\AIContextBuilder\` as well: it holds the design-time build cache.
2. Remove all entries whose target starts with `AIContextBuilder.` from the Windows Credential Manager. The API keys live **outside** the directory and are not removed with it.
3. If you changed `BasePath` or `DatabasePath`, delete the second location as well.
4. For every analyzed solution, delete `<solution folder>\<Name>.aicb.json` - and a possibly orphaned `<Name>.aicb.json.tmp`.
5. In every project where `aicb init` ran, edit the harness files by hand: remove the aicb entry from `.mcp.json` (and from `.codex/config.toml` or the `mcp` section of `opencode.json`), delete `.claude/skills/...` and `.agents/skills/...` and the guard file `.claude/hooks/aicb-symbol-guard.mjs` (or the Codex/OpenCode equivalents) and remove the aicb hook entry from `.claude/settings.json`, `.codex/hooks.json` or `opencode.json`.
6. Delete any exports you created yourself: context documents (`--output`), copied source files, usage reports, run templates, session exports, master-data exports and constellations.

If you work only with the CLI or the MCP server, you have **no** built-in way to delete the usage recording. Your only option would be to delete the `.acb` file - which also carries every session, every snapshot and every profile.

10.8 What the product does with your data

The network surface

Nothing leaves your machine except through an LLM action or armed desktop auto-initialization that you configure. The product's entire outgoing network capability consists of its three LLM clients. There is no update check, no version ping, no crash or error reporting, no analytics - and no other component that could transmit anything:

- The desktop application is the only part that owns an HTTP client. In the CLI and in the MCP server the capability is **absent**, not switched off: the MCP server composes no LLM client at all, so no code path exists that could open a connection.
- No client sends a credential over unencrypted HTTP to a remote host. All three refuse plain `http://` to a non-localhost address instead of sending. Plain HTTP to `localhost` remains allowed - that is Ollama or LM Studio on your own machine.

The LLM calls

There are exactly three entry points that can contact a model:

1. **A run** - triggered from the workspace.
2. **The connection test** - the `Test Connection` button in the model profile settings. It sends a ping of 16 tokens with a 20-second budget.
3. **First-load solution setup** - when `Auto-initialize via LLM on next load` is armed for Layer Profile or Exclude Namespaces, loading an unconfigured solution sends one request containing its declared and referenced namespace lists. The returned proposal is shown for confirmation before it is applied. Declining prevents the configuration change, but the request has already been sent. Freshly registered solutions start with these flags armed; clear them before the next Context Builder load if you do not want this call.

All three paths stop *before* a socket is opened if the selected profile is incomplete: the remote clients require their model and credential fields, while a local client requires a model name and endpoint.

The built-in model profiles come with pre-filled endpoints - Anthropic (`https://api.anthropic.com/v1/messages`), OpenAI (`https://api.openai.com/v1/chat/completions`), Google Gemini (`https://generativelanguage.googleapis.com/v1beta`), OpenRouter (`https://openrouter.ai/api/v1`), and the local ones Ollama (`http://localhost:11434/v1`) and LM Studio (`http://localhost:1234/v1`). A profile entry is a **template**: it carries an address but no key. Nothing is contacted until you select the profile, store a key and start a run or the connection test.

Process launches

The product starts a small number of external programs, and none of them is steered by the code it analyzes:

- `vswhere.exe` , at start-up, to locate an MSBuild the locator does not know.
- `dotnet apicompat` , only through the opt-in `compare_public_api` tool, with assembly paths you supply.
- `git rev-list` , from `server_info` , to report how far the running binary lags your checkout; the analyzed solution's folder is used as the working directory.
- In the desktop application: your default browser for the sponsor link, your default text editor for the license file, and Explorer on a selected solution.

The CLI and MCP process launches build argument lists without a shell and without concatenating analyzed file names, namespaces or symbols into command text. The desktop application's Explorer integration is a separate shell-executed exception: it interpolates the selected solution path into Explorer's `/select` argument.

The one caveat: MSBuild

Opening a solution runs its MSBuild build logic to resolve references and determine what to compile - exactly like Visual Studio or `dotnet build` . Building the compilation also runs the source generators the projects reference (generator packages, the .NET SDK's own generators and generator projects in the solution), inside the AICB

process, again as an IDE does. A deliberately malicious project could therefore run code at solution-open time. This is inherent to every MSBuild-based tool; the trust boundary is the same as "clone this repository and run `dotnet build`". **Only analyze solutions you trust.**

One thing the product does *not* do: run your solution's third-party Roslyn analyzers - with one opt-in exception for DiagnosticSuppressors, described below. The `get_diagnostics` tool returns compiler diagnostics only; diagnostics that an analyzer or a source generator reports itself are not part of it. To say where a real build would suppress some of them, it reads the analyzer assemblies' metadata only, which runs none of their code. The one exception is the opt-in argument `runSuppressors: true`: it loads exactly the analyzer assemblies that declare a DiagnosticSuppressor and runs those suppressors, as a real build does. Loading such an assembly also runs the constructors of every analyzer type it declares, inside the AICB process - the same trust boundary as the source generators above. AICB also does not index generated code: generated documents are not walked, and source-generated members such as `[ObservableProperty]` properties are re-derived from the attributes in the parsed syntax.

10.9 API keys

API keys are **not** stored in the database and **not** stored in any file the application writes. They are handed to the Windows Credential Manager, which stores them per user under the target name `AIContextBuilder.ApiKey:<model profile id>`. The application does not encrypt them itself; it stores them in the operating-system credential store, which is a better place than a database column and keeps them out of every backup and export.

Consequences you should know:

- The keys survive restarts and are stored per Windows user account.
- They are **not** part of the database, **not** part of a backup and **not** part of a constellation export.
- A constellation imported through the CLI or an MCP tool arrives **without keys**, because the headless composition has no access to the credential store. That is intentional - if a run then fails with a missing key, this is the reason.
- In the model profile panel, `Clear API Key` removes the stored key, and `Save API Key Only` persists a changed key without saving the other editor fields. Writing an empty secret is the same as deleting.

10.10 Local MCP usage recording

The MCP server records every tool call it handles into the table `tool_calls` of the **same database the GUI uses**. There is no separate telemetry database and nothing is transmitted: the only writer is the local database, and the only readers are the `usage_report` tool and the `MCP Usage` panel of the desktop application.

The recording is **on by default** as soon as the server starts with a usable configuration database - which is the standard case. The server names the database it uses on its error output at start-up: `No --db-path given; using the standard config DB (same as the GUI): <path>`.

What one row contains:

Field	Content
Timestamp	UTC time of the call, ISO 8601.
Tool name	The tool that was called, or `(unknown)`.

Field	Content
Session reference	A SHA-256 hash (the first 12 hex characters) of the session reference - never the `.sln` path. 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 for failures the exception **type** name.
Result size	The summed length of the returned text blocks.
Duration	The call duration in milliseconds.
Client name, client version, protocol version	What the MCP handshake reported; empty when a client does not identify itself.
Server version	The version of the server that recorded the row.
Alias applied	Which parameter alias was rewritten, if the server accepted an alternative parameter name.
Facet	The resolved task facet the call addressed, or the literal `(unresolved)` for an unknown spelling - never the caller's raw text.
Error message	The failing exception's message, capped at 500 characters. This is the **only** free-text field. When a failed call also sent arguments the tool does not have, the message ends with `[unrecognized arguments: <names>]` (at most three names, then a count of the rest).
Child calls	For `batch` / `measure`: a JSON tally of the sub-queries the call dispatched.
Solution size	Project count and file count 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` .

Never recorded: source code, argument *values*, or raw session paths. Argument names are recorded only where the tool does not have them. The recording is deliberately limited to shapes.

Reading it:

- The MCP tool `usage_report` returns the aggregate: 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 with error classes, latency and result-size percentiles. Note that its counts are a **lower bound**: a sub-query inside `batch` or `measure` is recorded on the parent row only, and a one-shot `aicb call` writes the same row under the client name `aicb-call` only when it is given `--db-path` - without one it records nothing.
- The `MCP Usage` panel shows the same data in the desktop application: `Refresh` , `Export` , `Import` and `Clear` at the top, and the regions `Tools` , `Calls` , `Notes` , `Errors` , `Clients & protocol` and `Snapshots` . In the `Calls` region you can attach a free-text note to an individual call; notes cascade with the call they

belong to.

Turning it off, in three steps:

1. **Completely off** - if the server starts without a usable configuration database, nothing is recorded at all. Note that this also costs the stored MCP profiles, so it is not a convenient switch.
2. **Free text off, shapes remain** - set the environment variable `AICB_MCP_ERROR_TEXT=off`. The error message is then not recorded; only the exception type name remains. This is meant for a configuration database that is shared rather than local.
3. **Delete afterwards** - the `Clear` button (dialog `Clear Usage Data`) on the `MCP Usage` page (desktop application only), as described under "Removing your data".

There is no opt-out per tool and none per MCP session.

10.11 Third-party components and licenses

AIContextBuilder is closed source and ships third-party libraries as compiled binaries. The product is distributed in three forms - as a `dotnet tool` NuGet package, as a Windows installer, and as a portable ZIP - and all three carry the file `THIRD-PARTY-NOTICES.txt`, which is the attribution those libraries require.

The list in that file is generated from the resolved dependency graph of the two projects that reach a user, so it is the set of components that actually ship. Its summary:

License	Components
MIT	72
Apache-2.0	6
BSD-3-Clause	1
Declared by URL (no SPDX expression)	1
Total listed	**80**

Of these 80 components, 76 ship a binary; 4 are metapackages or are supplied by the target framework and are listed because the product is built against them, not because they are redistributed. Each entry names the license, the copyright, the authors and the project, and states which distribution it reaches (CLI/MCP, GUI, or both).

The largest parts of that surface are the Roslyn compiler and workspace assemblies (which do the analysis), the MSBuild integration, the MCP SDK (Apache-2.0), and SQLite - including the **native** SQLite binaries for many platforms, which is why the attribution covers machine code and not only managed assemblies. The file also carries the copyright notices and the license texts of the components that require them.

The product's own license is separate and sits beside the notices file as the full bilingual `EULA.md`, with the non-binding orientation summary in `LICENSE.txt`.

10.12 What the product promises and requires

The license terms (the three thresholds, the always-free cases, the restrictions and the liability rules) are described in "License, installation and updates"; the prerequisites and the one-installation-per-machine rule are described there as well.

One statement belongs here, because it is about data: the 90-day transition period is contractual text only, not a timer in the product. The software transmits no threshold or deadline data, blocks no feature and contains no license server, activation token, watermark mechanism, phone-home or outbound licensing telemetry. This does not mean that nothing is recorded locally: the MCP usage table described above records tool calls for local inspection and never leaves your machine.

Questions, bugs and feature requests go to the project's GitHub Issues; please include `aicb --version` and - for the MCP server - the output of the `server_info` tool. Security issues go through the private route described in `SECURITY.md` (email `aicb@dadera.de`, or GitHub's private "Report a vulnerability").

11 Troubleshooting

This chapter lists the problems you are most likely to meet, what causes them, and what to do about them. Each entry names the symptom exactly as it appears on screen, because the fastest way to use the chapter is to search it for the message you are looking at.

Two areas have their own manuals: problems that belong to the MCP server (tool pools, sessions, drift) are described in the MCP manual, and problems that belong to the desktop application's panels and editors are described in the desktop application manual. The entries below are the ones that apply to the product as a whole.

11.1 Startup and installation

`MSBuild not found` - the application does not start

Symptom. A dialog titled `MSBuild not found` appears, and after you close it the application exits with exit code `1`.

```
AIContextBuilder could not find a compatible MSBuild installation.

Tried:
  - MSBuildLocator.RegisterDefaults
  - MSBuildLocator.QueryVisualStudioInstances
  - vswhere.exe -> VS\MSBuild\Current\Bin
  - Newest .NET SDK in %ProgramFiles%\dotnet\sdk

Possible causes:
  - Visual Studio is installed, but the '.NET desktop development' or 'MSBuild Tools' workload is
missing.
  - Neither VS nor a .NET SDK is installed.

Diagnostics (in a PowerShell):
  & "C:\Program Files (x86)\Microsoft Visual Studio\Installer\vswhere.exe" -latest -products * -
requires Microsoft.Component.MSBuild -property installationPath
  dotnet --list-sdks

The application will exit.
```

Cause. To load a C# solution, the application needs an MSBuild installation, which comes either with a .NET SDK or with Visual Studio. It tries four routes in this order: the locator's default discovery, the newest Visual Studio instance the locator knows, `vswhere.exe`, and the newest .NET SDK under `%ProgramFiles%\dotnet\sdk`. The last three are Windows-only. If all four fail, there is no analysis workspace and the application cannot continue.

What to do. Install a .NET **SDK** (not just the runtime), or Visual Studio with the `.NET desktop development` or `MSBuild Tools` workload. The two commands printed in the dialog tell you what the machine currently has:

Command	What it shows
`& "C:\Program Files (x86)\Microsoft Visual Studio\Installer\vswhere.exe" -latest -products * -requires Microsoft.Component.MSBuild -property installationPath`	the installation path of the newest Visual Studio that has the MSBuild component
`dotnet --list-sdks`	every .NET SDK installed on the machine

Note: a self-contained installation removes the dependency on the .NET **runtime**, not the dependency on MSBuild. See "License, installation and updates".

The same problem on the command line

`aicb analyze` stops with exit code `5` and writes to stderr:

```
error: MSBuild registration failed on <platform>: <reason>. aicb runs on .NET <N> here and cannot use a .NET SDK that is newer than that. Install the .NET <N> SDK and check that 'dotnet --list-sdks' lists it. The .NET tool can instead run on the newest installed .NET: set DOTNET_ROLL_FORWARD=LatestMajor, and DOTNET_ROLL_FORWARD_TO_PRERELEASE=1 as well if that .NET is a preview. On Windows a global.json next to the .sln can help disambiguate SDK versions.
```

The CLI uses only the first of the four routes, so on Windows the message points at the SDK rather than at Visual Studio.

`aicb mcp` and `aicb call` do **not** stop. `aicb mcp` starts with this warning on stderr:

```
warning: MSBuild could not be registered (<reason>). analyze_solution/refresh_session will fail; the remaining tools stay usable.
```

`aicb call` uses a different warning because it can invoke session-less tools directly:

```
warning: MSBuild could not be registered (<reason>). Tools that analyze a solution (self-init / analyze_solution) will fail; others still run.
```

That warning is a prediction, not a startup failure: the tools that need a loaded solution (`analyze_solution`, `refresh_session`) fail later, while database-bound queries, `server_info` and `docs` keep working. The remedy is the same as above.

Do you need the .NET runtime?

Installation form	Needs the .NET runtime?
Setup installer (Windows)	No - self-contained, the runtime is included
ZIP archive (Windows)	No - self-contained
`dotnet tool install -g aicb-roslyn-mcp`	Yes - a .NET tool is framework-dependent and needs the .NET 10 SDK, which is also what analyzing a solution needs

Windows SmartScreen warns about the download

Symptom. On first start of the installer or of the unpacked `aicb-ui.exe`, Windows shows "Windows protected your PC" and blocks the file.

Cause. SmartScreen warns about a file that does not yet have enough download history. From 0.5.503.1 on the files are digitally signed by `Gregor Dadera`, so the warning names that publisher, but a new release can still trigger it until it has been downloaded often enough. Versions up to 0.5.502.1 were not signed at all, and the warning showed an unknown publisher.

What to do. Choose `More info` and then `Run anyway`. The same instruction is in `LIESMICH.txt` next to the unpacked files and in the README of the public repository. To make sure the file is genuine, check `Properties > Digital Signatures` for the signer `Gregor Dadera` (see "The Windows SmartScreen warning"); every release also lists SHA-256 checksums for its files.

`Database is newer than this build`

Symptom. A dialog titled `Database is newer than this build` appears and the application exits with exit code `2`.

```
This database was written by a newer version of AIContextBuilder.

Database schema: <N>
This build knows: <M>

The application will exit rather than write to it, because an older
build would silently save the outdated shape back. Update
AIContextBuilder, or point the database path in the app settings
(Storage) at a different file.
```

Cause. The database was written by a newer version than the one you just started. The usual way this happens is two installations side by side: an unpacked ZIP that is newer than the installed version was started once and upgraded the shared database under `%APPDATA%\AIContextBuilder\`. Schema upgrades are one-way; a newer database cannot be downgraded.

What to do. Either bring the older installation to the same version, or choose a different database path under `Settings > Storage`. A newly chosen path starts with an empty database. See also "Data, storage and privacy".

Note: only the desktop application refuses to start. The MCP server tolerates a newer database and reports the difference through `server_info`; the CLI verbs neither refuse nor report it.

Host	Behavior with a database that is newer than the build
Desktop application	Dialog `Database is newer than this build`, exit code `2`
MCP server (`aicb mcp`)	Starts anyway; reports the drift through `server_info`
CLI verbs (`aicb analyze`, `aicb export`, ...)	Neither refuses nor reports

The same check applies when you switch databases in the desktop application: `Target DB schema version <N> is newer than this app's latest (<M>). Upgrade the app or pick a different DB.`

`Database migration failed`

Symptom. A dialog titled `Database migration failed` appears and the application exits with exit code `2`.

```
AIContextBuilder could not update the database schema.

Error: <message>

The application will exit. Verify that the database path in the
app settings (Storage) is reachable and writable.
```

Cause. The schema migration could not be completed - for example the path is unreachable, the file is read-only, the file is not a valid database, or a migration committed and the follow-up foreign-key check found violations:

```
Schema migration v<N> committed but PRAGMA foreign_key_check reported <K> violation(s). Inspect the
DB and decide whether to delete the offending rows or restore missing parent rows. First few rows:
<table#rowid->parent>; ...
```

What to do. Check the database path under `Settings > Storage` and make sure the folder is reachable and writable. If another program (typically a backup tool) is holding the file, close it and start the application again. If the message reports foreign-key violations, keep a copy of the database and contact support (see the Support section below).

`'...' is not an AIContextBuilder database`

Symptom. When you switch or create a database, or start with a manually edited path:

```
'<path>' is not an AIContextBuilder database: the file must end in '.acb'. Rename it (and point
Storage.DatabasePath at the renamed file), or pick a '.acb' file.
```

Cause. The application's own database must end in `.acb`; the default is `aicb.acb` under `<BasePath>\user-data`. The rule exists so that one SQLite file cannot be used under two names by two parts of the product, which would look like data that comes and goes.

What to do. Rename the file so that it ends in `.acb` and point `Storage.DatabasePath` in `%APPDATA%\AIContextBuilder\app-settings.json` at the renamed file - or pick an existing `.acb` file in the dialog. The file dialog deliberately offers only `.acb`.

Note: a database path that you pass explicitly to a CLI verb or to an MCP tool is not checked against this rule; there you are naming a specific existing file on purpose.

The settings file could not be read

Symptom. The application starts normally, but the database path, the base path and the lists of recently opened solutions and databases are back at their defaults, and a notice bar appears at the top of the window:

```
Application settings - Your settings file could not be read, so this session is running on DEFAULTS - including the database path, which is why your sessions and solutions may look missing (<reason>). A copy of it is preserved at <path>.
```

Cause. `%APPDATA%\AIContextBuilder\app-settings.json` could not be parsed (or could not be opened at all). Because that file also carries the database path, the application then runs against the default database - which is why your sessions, solutions and runs appear to be gone. They are not: only the pointer to them was lost.

What to do.

1. Open `%APPDATA%\AIContextBuilder\app-settings.json` and restore the `Storage.DatabasePath` value, or select the old database again under `Settings > Storage`.
2. The unreadable file is preserved next to itself as `app-settings.json.corrupt-<timestamp>`. If the notice says that no copy could be made, copy the file elsewhere before you open a solution, because opening a solution rewrites the settings file from the defaults.

If the file could not even be opened, the notice reads `Application settings - Your settings file could not be opened (<reason>), so this session may be running on DEFAULTS - including the database path.`

The start takes a long time

Symptom. Several seconds pass after you start the application before a window appears.

Cause. The windowless part of the startup does the work before the main window exists; a splash screen is shown on its own thread while it runs. The one case that can turn seconds into a long wait is the automatic backup: if it is enabled (default: off) and its interval has elapsed, it runs synchronously before any window is created. With a large base path and a large database this can add tens of seconds. A **failed** backup run does not advance its timestamp, so the delay repeats on every start until the cause is fixed.

What to do. Wait for the window. If the delay repeats, read the startup notice bar - it names the reason (see "Automatic backup reports"). If you do not need the automatic backup, switch it off under `Settings > Storage`.

`Unfinished runs from previous session`

Symptom. At startup a dialog appears with three buttons:

```
During the previous app session, <N> run(s) did not finish cleanly:

  - <name> (<type>, started <yyyy-MM-dd HH:mm>)
  ... and N more

What should happen with them?
  Yes       = Mark as Cancelled (close cleanly)
  No        = Pause for later resume
  Cancel    = Leave unchanged (decide later)
```

Cause. Runs that were still marked as running when the previous session ended - typically after a crash, a hard kill or a power loss.

What to do. `No` is the least lossy choice if you still care about the run: it is paused so you can resume it later. `Yes` closes it cleanly as Cancelled. `Cancel` (or closing the dialog) leaves everything as it is and asks again at the next start. The list shows at most five entries; the rest are summarized as `... and N more`.

If the recovery itself fails, a warning titled `Crash recovery` appears and the application starts anyway:

```
Crash recovery failed.

Error: <message>

The app will start anyway; unfinished runs remain with status=Running.
```

Note: `Iteration` and `Preselection` run templates are not yet released - only `Manual` templates can be selected. The recovery dialog concerns runs of any type that were left running.

11.2 Analyzing a solution

`error: solution not found` / `Solution file not found`

Symptom (CLI). `error: solution not found: <path>` on stderr, exit code `1`.

Symptom (desktop application). A dialog titled `Open solution`:

```
Solution file not found:
<path>
```

Cause. The given path does not exist. The desktop application shows this dialog only when the solution is unknown to it; a known solution whose file has moved offers to relocate it instead (next entry).

What to do. Check the path, the drive letter and the file name. If the solution was moved, see the next entry.

The solution has moved

Symptom. When you open a known solution or a stored session, a relocate dialog appears. It shows the old path and asks you to pick the new location.

Cause. The stored solution path no longer exists - typical when a database was carried to another machine, or after a folder was renamed.

What to do. Point the dialog at the solution's new location and confirm. The application updates its record and continues opening. The same dialog is used when you open a stored session whose solution has moved.

Follow-up messages you may see:

Message	Title
`Solution file still not accessible at:\n<path>`	`Open solution`
the error of the failed database update	`Relocate solution`
`Session no longer exists.`	`Open Session`

Message	Title
`The associated solution could not be found.`	`Open Session`

The solution was never built - the most important case

Symptom. Answers come back and look plausible, but they are too small. Typical examples:

- A side-effect query (`find_by_side_effects`) answers `count: 0` while its own `methodsScanned` counter reports thousands of methods - which reads like a passed architecture check.
- An external-call query (`calls_external`) answers zero for an API the solution obviously uses, for example `System.IO.File` in a solution whose job is writing files.
- `get_diagnostics` lists every project under `incompleteProjects` .

Cause. Roslyn cannot resolve the projects' references: no `obj/` folder, no NuGet restore, or a missing targeting pack for the target framework. Unresolved code still parses, so facts are produced - but without the call edges and type identities the answers are derived from. Nothing looks broken; the numbers are simply wrong.

How the application tells you. Every affected answer carries an alarm that qualifies the **run**, not the individual answer - so it also appears over a healthy-looking, non-empty list:

```
THIS ANALYSIS RUN WAS INCOMPLETE: <k> of <n> scanned projects could not resolve their references
(<names>, and <r> more). <consequence> Restore/build the solution, then refresh_session and re-run
this query.
```

The middle sentence is worded per question family, because something different breaks in each case:

Question family	What the alarm tells you
Fan-in, dead code, instantiation	The numbers are short by an unknown number of **ordinary** references, not merely exotic ones
Side effects and external calls	Effects are derived by propagating along call edges, so a method whose calls did not resolve is reported as **pure** - treat the result as a floor, never as a passed purity or clean-architecture check
Resource leaks	Both the leak list and the count of creations checked are short - treat a zero here as unmeasured, never as leak-free code
XAML bindings	Breaks in **both** directions: an unresolved scope is silently skipped (under-reporting), while a view model whose generated members did not bind looks checkable but is not (false reports)
Event subscriptions	A subscription is recorded only where the left-hand side resolved to an event, so both the matches and the `availableEvents` list are short - treat an empty list as unmeasured, never as an absence
Code traits, interface and base-type relationships	The resolved traits (`reflection` , `linq-in-loop`) and the declaration relationships can be missing or attached to the wrong candidate - treat them as partial evidence

If a query's scope matched nothing at all, the answer says so instead:

```
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.

1. Build the solution once, from the command line in the solution's directory:

```
dotnet restore
dotnet build
```

2. Tell the session about it: call `refresh_session` (MCP), or load the solution again in the desktop application.
3. Repeat the query. `get_diagnostics` is the fastest check that the run is now complete: `incompleteProjects` should be empty.

Note: a freshly cloned repository is the most common case. If the solution cannot be built on your machine (a missing targeting pack, for example), the alarm stays - and the answers stay marked as incomplete, which is exactly what the marking is for.

Note: a solution that defines no `Any CPU` platform can stay incomplete after a build for another reason: the design-time build ran at a platform the build did not write. `get_diagnostics` names the platform used in its `designTimeBuildPlatform` field; set `designTimeBuildPlatform` in the solution's `.aicb.json` to the platform you build at, then call `refresh_session(force: true)` (see "The build platform").

Note: two further causes leave errors about names that do not exist, and `get_diagnostics` names both:

- **A git submodule that was never initialized.** Its folder is empty, so the sources a project's build files name inside it are missing from the compilation, and neither a restore nor a build brings them. `uninitializedSubmodules` lists each such submodule in its `submodules` list (`path` , the `projects` that reach into it, their `unresolvedNameErrors`), and its `note` gives the repair: in the repository root run `git submodule update --init <paths>` , then `refresh_session(force: true)` .
- **A compile item without a file.** A project lists a file that does not exist on disk - typically one a pre-build step or a code generator writes, which did not run before the analysis. `missingCompileItems` names the project file, the missing files and the errors in that project. Run the step that writes the files, then `refresh_session(force: true)` ; if nothing writes them, the project lists a file that does not exist.

What `incompleteProjects` means for a number

Symptom. `get_diagnostics` reports `errorCount` - possibly a large one, possibly a suspiciously small one - next to `incompleteProjects` , `incompleteRatio` and `suppressedDiagnosticsTotal` .

Cause and meaning. A project whose core references do not resolve (for example `System.Object` is missing because a targeting pack is not installed, or the project is not restored) would produce thousands of cascade errors (`CS0518` , `CS0234` , `CS0246`). Such projects are therefore not listed as diagnostics; they are disclosed separately under `incompleteProjects` , each with the project name and the number of diagnostics that were suppressed with it. The counts around them mean different things:

Field	Meaning
`errorCount`	Errors only, deduplicated across target frameworks

Field	Meaning
`suppressedDiagnosticsTotal`	Everything from the chosen severity floor upward that the incomplete projects contributed, **not** deduplicated - a multi-targeted project counts once per target framework. Read it as a magnitude, not as one half of a ratio with `errorCount`: a total that dwarfs `errorCount + warningCount + infoCount` means that only a fraction of the solution was measured
`cascadeClassifiedCount`	Diagnostics from projects that **do** resolve but inherit broken references from an incomplete project. They are never filtered away; they are marked `cascadeFromIncomplete: true`, and each `incompleteProjects` entry names its `affectedDependents` (transitively)
`referenceBindingAdvisoriesSuppressed`	`CS1701` / `CS1702` assembly-version advisories without a source location; they cannot be fixed from source and are filtered out
`verdict: 'inconclusive'`, `reliable: false`	Set only when a strict majority of the scanned projects are incomplete. The disclosure (`incompleteRatio`, `suppressedDiagnosticsTotal`) is not rationed the same way: it appears as soon as a single project is incomplete
`projectsScanned`	The denominator: how many projects were scanned

What to do. Build the solution, then `refresh_session`. Only after that is `errorCount` a statement about the solution.

Load problems that are not shown in a dialog

Symptom. Symbols are missing from answers, and no dialog appears.

Cause. Problems while loading a project (an unsupported project type, a broken project file, a missing target) are logged as `[Workspace] <kind>: <message>` rather than shown in the interface. In the desktop application they land in `%APPDATA%\AIContextBuilder\aicb.log`; in the MCP server they go to stderr. A project of an unsupported type therefore drops out of the analysis without a visible message.

What to do. Check `aicb.log` (desktop application) or the stderr log of your MCP client for `[Workspace]` entries; make sure the project type is one Roslyn can load; restore and build the solution.

The analysis is cancelled

Symptom (CLI). `cancelled.` on stderr, exit code `3`.

Cause. You pressed Ctrl+C, or the operation was cancelled. This is the intended way to stop an analysis; the workspace is cleaned up.

What to do. Run the command again when you are ready.

Two analyses of the same solution at the same time

Symptom. A call hangs for a while 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 session initializes itself from a `.sln` path, the same 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 MCP server gives up after 45 seconds by default, so that you learn what is holding the gate instead of waiting silently.

What to do. Wait - do not retry, because a retry queues behind the same operation again. For very large solutions, analyze a solution filter (`.slnf`) instead of the whole solution.

Note: the wait can be changed for the MCP server with the environment variable `AICB_MCP_LOAD_WAIT_MS` (milliseconds). The desktop application and the CLI wait without a limit.

11.3 LLM and network errors

The desktop application can send a generated context document to an LLM: Anthropic, OpenAI, or `Custom` for any local OpenAI-compatible server (Ollama, llama-server, LM Studio, vLLM, and others). Errors are returned as a message on the result, not as a crash. For HTTP errors the provider's own response body is included, truncated to 800 characters.

The message catalog

Situation	Message
Model name missing	`Anthropic: ModelName is required.` / `OpenAI: ModelName is required.` / `Local: ModelName is required.`
API key missing	`Anthropic: ApiKey is required.` / `OpenAI: ApiKey is required.`
Endpoint missing (local)	`Local: Endpoint is required (e.g. http://localhost:11434/v1).`
Key would be sent in cleartext	`Anthropic: API key not sent over plain HTTP to a non-localhost host. Use an HTTPS endpoint.` (the same for OpenAI; local: `Local: bearer token not sent over plain HTTP to a non-localhost host. Use an HTTPS endpoint or omit the API key for unauthenticated local servers.`)
HTTP error status	`Anthropic HTTP <code>: <body>` / `OpenAI HTTP <code>: <body>` / `Local HTTP <code>: <body>`
Timeout	`<provider> timeout: HttpClient default timeout exceeded.` (the HTTP client timeout is 5 minutes)
Network error	`<provider> network error: <message>`
Other error before the stream starts	`<provider> unexpected error: <message>`

Situation	Message
You cancelled the call	`Cancelled.`
Response could not be read	`Response parse error: <message>` (Anthropic: `Anthropic response parse error: <message>`)
Unknown provider	`Unsupported provider '<x>'. Supported: Anthropic, OpenAI, Custom (local OpenAI-API-compatible: Ollama / llama-server / LM Studio / vLLM / etc.).`
No result recorded	`LlmClientDispatcher: no response captured.`

For `HTTP 401` and `HTTP 403` the provider's body tells you whether the key is wrong or the account has no credit; these statuses are not retried.

Note on local endpoints: enter the base URL of your server (for example `http://localhost:11434/v1`). A bare host is completed to `/v1/chat/completions`, an already versioned base (for example `/v1beta`) gets `/chat/completions`, and a full chat-completions path is accepted unchanged.

A truncated answer

Symptom.

```
Stream ended unexpectedly (no finish_reason/[DONE] received; response may be truncated).
```

For Anthropic the wording is `Anthropic stream ended unexpectedly (no stop_reason/message_stop received; response may be truncated).` The partial text received so far is kept and delivered together with the message.

Cause. A healthy stream ends with a finish marker. If the marker is missing, the connection was closed before the model finished - a clean end of the connection is not proof that the answer is complete.

What to do. Repeat the run. If it happens repeatedly, check a proxy or firewall between you and the provider, or - for a local server - whether the server ran out of memory.

Rate limits and automatic retries

The application retries a failed call automatically for these status codes: `429`, `500`, `502`, `503`, `504`, and for network and timeout errors. If the provider sends a `Retry-After` header, that value is used (capped at the maximum backoff); otherwise the wait grows exponentially from the base backoff.

Setting (`Settings > General`, section `LLM Retry`)	Default	Meaning
`Max attempts`	`3`	Total attempts per call; `1` means no retry
`Base backoff (seconds)`	`1`	First retry delay; doubled on each further attempt (1s → 2s → 4s ...)
`Max backoff (seconds)`	`30`	Upper bound for a wait, including a provider `Retry-After` value

Changes take effect after an application restart. A cancellation by you is never retried, and neither is `401` / `403` - retrying a wrong key would only repeat it.

Before a run: cost estimate and context window

A single `Manual` call starts without a cost pre-flight. For the `Iteration` and `Preselection` run types - **not yet released**, see the note below - the application estimates the token budget before sending and asks for confirmation when the estimate reaches the configured threshold:

Setting (`Settings > General` **, section** `Run Confirmation` **)**	**Default**
`Confirm threshold (tokens)`	`50000`

Note: the comparison is "greater than or equal", so a threshold of `0` means the application asks before **every** run, not never.

Independent of the run type, a request whose input plus maximum response would exceed the model's context window triggers a warning before sending, with the estimated size and the model's window. Lower the token budget or pick a model with a larger window.

Note: `Iteration` and `Preselection` run templates are wired but not finished, so they are not released yet; only `Manual` templates can be selected. `Pipeline` exists in the stored data model only and executes nothing.

11.4 Data and persistence

Switching or creating a database fails

All messages from `Settings > Storage` appear as the error of the operation:

Situation	Message
Empty path	`Database path is empty.`
Wrong extension	see `'...' is not an AIContextBuilder database` above
Schema version unreadable	`Could not read schema version: <message>`
Target database is newer	`Target DB schema version <N> is newer than this app's latest (<M>). Upgrade the app or pick a different DB.`
Target needs migrations	`Target DB needs <N> migration(s); confirm to apply them.`
Background work running	`Cannot switch DB while <k> background task(s) are active.` / `Cannot create DB while <k> background task(s) are active.`
Activity slot not available	`Could not reserve DB-Swap activity slot: <message>`
Settings file could not be written before the swap	`JSON write failed before swap: <message>`
Migration failed, rollback succeeded	`Migration failed: <message>. JSON rolled back to previous path.`
Migration failed, no previous path	`Migration failed: <message>. No previous path to roll back to - JSON points to the new (broken) target.`

Situation	Message
Rollback failed as well	appended to the previous message: `Rollback itself failed: <message>.`
Swap succeeded, cache refresh incomplete	`Swap succeeded but cache-invalidation partially failed: <message>. Restart the app to refresh stale caches.`
Creating a database where a file already exists	`File already exists: <path>. Use Switch for an existing DB.`
Directory could not be created	`Could not create directory: <message>`

Each message also tells you what state the settings file is in afterwards - whether it was rolled back to the previous path, whether there was no previous path, or whether the rollback itself failed. That is the information you need for the next step.

A stored snapshot is not reused

Symptom. An analysis that normally reuses a stored snapshot runs fully instead, without an error.

Cause. The stored snapshot no longer matches the running build - the persisted format or the identity of the analysis engine changed (typically after an update) - or the file-set hash could not be computed reliably. The application discards the snapshot and analyzes fully rather than risk loading an incompatible one.

What to do. Nothing; the full analysis is the correct reaction. The only thing worth knowing is why an analysis takes longer once after an update.

`attempt to write a readonly database`

Symptom. This message appears while the application is writing to the database.

Cause. A second process holds the database file: a running MCP server instance, a second desktop application instance, or a backup program. When the holder is an archiver, SQLite's open succeeds but is silently downgraded to read-only, so every write fails with this message.

What to do. Check whether a second instance of the application or an `aicb mcp` process is running (an MCP server that an agent keeps alive is the usual case), close it if it is not needed, and restart the application.

Automatic backup reports

The automatic backup reports when something is missing or failed: `Partial` and `Failed` are shown in the startup notice bar, prefixed with `Automatic backup at startup -`. It also reports a successfully written archive when the last-backup timestamp could not be stored. A clean backup with successful bookkeeping, and a start on which none was due, stay silent.

Situation	Message
Base path does not exist	`BasePath does not exist: '<path>'.`
Backup folder cannot be created	`Could not create backup folder: <message>`
Backup failed	`Backup failed: <message>`

Situation	Message
No free file name	`Could not find a free backup file name in '<folder>' after 100 attempts.`
Archive written	`Backup written to <path>`
... with locked files	`(skipped <k> locked file(s): <names> and <r> more)`
... without the database	`(database NOT archived: <reason>)` - for example `path unusable - <message>`
... with a failed timestamp write	`The archive was written, but the backup timestamp could not be stored (<reason>) - so this backup will run again at the next start.` (or, with the backup switched off: `... the last-backup time still shows the previous run.`)

Note: an incomplete archive is renamed with a `-partial` marker before the `.zip` extension, and the newest complete archive is never removed by the retention rule - so a series of partial backups cannot evict the last archive that actually contains your database.

The bar is dismissible, and the dismissal is deliberately not remembered: a problem that is still there reports again at the next start.

A note cannot be saved

Symptom. In the `MCP Usage` panel:

```
The note could not be saved: the call it belongs to is no longer in the database. It was probably cleared in the meantime.
```

For any other error the message is `The note could not be written: <message>`.

Cause. The tool call the note belongs to was deleted in the meantime, so the note's foreign key no longer resolves. The two texts are split on purpose: only this case gets the explanation, while every other failure (a locked file, a wrong database path) carries the real error message.

What to do. For the first case there is nothing to do - the call is gone. For the second, read the message: it names the actual cause.

An imported usage report seems to be missing

Symptom. You imported an MCP usage report, but the numbers in the `MCP Usage` panel do not change.

Cause. An imported report is stored **beside** the live data, as a named snapshot - it appears in the `Snapshots` region of the panel, not in the live figures above it, which always aggregate only the local calls.

What to do. Switch to the `Snapshots` region. When there are no local calls but an imported snapshot exists, the panel selects that region for you.

11.5 Diagnostic means

Means	Where	What it shows
`aicb.log`	`%APPDATA%\AIContextBuilder\aicb.log` (desktop application only)	Warnings and errors of the desktop application; rotates at 1 MiB, with `aicb.log.1` to `aicb.log.3` as archives; the file can be copied while the application runs
`load-perf.log`	`%APPDATA%\AIContextBuilder\load-perf.log` (desktop application only)	Solution-load phase timings, no errors; enabled with `Settings > General` → `Record solution-load performance timings`; does not rotate
`app-settings.json`	`%APPDATA%\AIContextBuilder\app-settings.json`	Storage paths and the recent lists
`server_info`	MCP tool, or `aicb call server_info`	Version, build commit, config-database schema version, and drift between binary, config database and the analyzed repository
Staleness remark	attached automatically to affected MCP answers	whether the answer came from a graph older than the files on disk; see "Sessions and staleness" in the MCP manual
`THIS ANALYSIS RUN WAS INCOMPLETE`	attached automatically to affected MCP answers	how many projects could not resolve their references (see "The solution was never built")
`get_diagnostics`	MCP tool	Compiler errors and warnings in seconds instead of a full `dotnet build`
`list_mcp_profiles` / `list_skills`	MCP tools	which tools the active profile exposes, and which exist outside it
`usage_report` / `MCP Usage` panel	MCP tool / desktop application	what the server actually did, per tool call
stderr of the MCP client	client-dependent	startup warnings, Roslyn load warnings, fatal background errors
CLI exit codes	shell	see the table below

Exit code	Meaning
`0`	Success
`1`	User error (invalid arguments, missing files)
`2`	Runtime error (unexpected exception)
`3`	Cancelled (Ctrl+C)
`4`	Migration failed

Exit code	Meaning
`5`	MSBuild not found
`6`	Quality gate failed (`analyze --fail-on` ; the document is still produced)

The CLI and the MCP server write no log file of their own; their channel is stderr.

Environment variables for the MCP server process (invalid or non-positive values fall back to the built-in default):

Variable	Effect
`AICB_MCP_TOOLS`	Tool curation for the process
`AICB_MCP_SESSION`	Comma-separated `key=value` list with the keys `ttlMinutes` , `maxSessions` , `sweepMinutes` (for example `maxSessions=4,ttlMinutes=120`)
`AICB_MCP_AUTO_REFRESH`	Auto-refresh mode. Built-in and newly created profiles seed `Reactive` ; without a resolved configuration database/profile, or when the field is absent, the effective mode is `Off`
`AICB_MCP_DEBOUNCE_MS`	Debounce window of the file watcher
`AICB_MCP_STALENESS_WINDOW_MS`	Throttle window of the staleness check
`AICB_MCP_LOAD_WAIT_MS`	How long a tool call waits at the load gate (see "Two analyses of the same solution at the same time")
`AICB_MCP_ERROR_TEXT=off`	Do not write the free text of an error message into the usage record; only the exception type is kept

Which version am I running?

- Desktop application: `Settings > About` .
- CLI and MCP server: `aicb --version` ; the MCP server additionally reports its version and build commit through `server_info` .

Note: if you installed both the installer and the .NET tool, two copies share one data folder. Install one of them per machine - see "License, installation and updates".

If the application reports an unexpected error

A dialog titled `Unexpected error` with this text means that an error reached the outermost handler:

```
An unexpected error occurred.

The application is still running, but its state may be inconsistent. Save any open sessions and
restart.

Details: <Type>: <message>
```

The `Details:` line names the exception type and message, including up to three nested levels. The application stays alive on purpose, so that you can save open sessions before you restart. At most three such dialogs appear per session; further occurrences of the same failure are written to `aicb.log` without a dialog, so that file is where to look if the interface keeps misbehaving quietly.

What to include in a report

- `%APPDATA%\AIContextBuilder\aicb.log` together with the archives `aicb.log.1` to `aicb.log.3` (desktop application).
- The version you are running (see above).
- For an analysis problem: the name of the solution - not its content.
- For a dialog: a screenshot, including the `Details:` line if it has one.
- `%APPDATA%\AIContextBuilder\app-settings.json` if the problem concerns paths or a database.
- For an MCP problem: the stderr log of your MCP client.

11.6 Support

- Questions, bug reports and feature requests: **GitHub Issues** at `github.com/gregordadera/aicb-roslyn-mcp/issues`.
- Security reports: please do **not** open a public issue. Send them to `aicb@dadera.de` or use **Report a vulnerability** on the `Security` tab of the repository. You will get an acknowledgment; there is no response-time deadline.

12 Glossary

This glossary explains the terms this manual uses. UI labels, tool names, parameters, environment variables and file names are set in `code` and quoted exactly as they appear on screen or in a file. Where a term carries more than one meaning in the product, the entry says so and gives the qualified form to use. Terms marked **not yet released** exist only as placeholders in the data format and cannot be used yet.

12.1 A - B

`aicb.acb` - The single application database (SQLite). It holds the master data, the known solutions, sessions, snapshots, runs and the local MCP usage record. The file name must end in `.acb`; any other extension is refused, both when opening and when creating a database. Its location is the path set under `Settings > Storage` (default `{BasePath}\user-data\aicb.acb`). See the file table under "Terms that are easy to confuse".

`aicb.mcp.json` - The **server** configuration of the MCP for headless operation, read once at server start. It sits with the server data (default `<ApplicationData>/AIContextBuilder/aicb.mcp.json`) and sets the exposed tool set and the session cache. Precedence: defaults < `aicb.mcp.json` < environment variable (`AICB_MCP_TOOLS`, `AICB_MCP_SESSION`).

`.aicb.json` - The **sidecar** of a solution: a git-tracked JSON file next to the `.sln`, named `<SolutionName>.aicb.json`. It can carry layer rules and strictness (`layeringPolicy`), namespace exclusions, test-project rules and test-attribute names, suppressions, auto-init flags, the preferred-target-framework scope and the design-time build platform. Consumers apply these fields differently: database-free headless analysis uses the layer/exclusion fallback, the analysis scope and the build platform, but does not resolve its test profile from the sidecar; suppression-aware reading tools separately combine their documented database and sidecar sources. The desktop app restores profile content into its database. The file carries configuration only - no analysis results, sessions, snapshots or credentials. `Export Config` on the `Workspace` page and the MCP tool `apply_solution_config` write it; `Import Config` and `solution_config_status` read it. `aicb init` does not touch it.

`.mcp.json` - The **client** configuration of your agent tool: it tells the agent which server to start. `aicb init` can write an entry for it, but the file belongs to the agent, not to aicb. Not to be confused with `aicb.mcp.json`.

Agent harness - The environment your AI agent runs in and that has its own configuration: Claude Code, Codex, OpenCode. `aicb init` detects it and can write skills and a symbol guard into it. `aicb init --hooks` accepts `auto` (default), `none`, `all`, `claude-code`, `codex` or `opencode`.

Analysis - The read-only pass in which aicb loads a solution with Roslyn and derives its symbol graph and facts. Without an analysis there is nothing to export and no session-bound MCP tool can answer; session-less tools such as `server_info`, `list_skills`, `usage_report`, `docs` and `list_mcp_profiles` remain available. The analysis never modifies your source files.

Auto-refresh - How the MCP server keeps a session's analysis in step with the code on disk. Three modes: `Off` (nothing happens by itself; a drifted session is disclosed and repaired only when `refresh_session` is called), `Reactive` (before a reading tool answers, the server re-analyzes if needed - the shipped default), `Proactive` (a file watcher starts the re-analysis once saving has gone quiet). The mode is set per MCP profile on the `Session` sub-tab.

Axis - In this manual, one of the three per-solution configuration axes: layer mapping, namespace exclusions and test detection. Other dimensions have their own names (detail level, severity, facet).

`batch` - An MCP tool that runs several **read-only** queries in one round trip. A sub-query that fails does not take the others down, and a result too large to fit is omitted with a pointer to the direct call. Not every tool can be batched: a tool that mutates, holds state, recurses or is otherwise unsuitable is refused, and the refusal names the reason. At most 16 sub-queries per call.

Built-in - A master entry that ships with the product. A built-in can be edited; editing marks it `Overridden` and keeps the original, which `Restore Built-In` brings back.

Bundle - An opt-in group of tools outside the task facets. The product ships one, `api-surface` (the public-API compatibility tools). Enable it per MCP profile or through `AICB_MCP_TOOLS`.

12.2 C

Card - A single entry in a list, for example in a master-data list or the snapshot list.

Chip - A clickable filter or switch: the five layout-mode chips in the Context Builder header, or the severity filters in the `Insights` tab. A non-clickable marker on a card (`Built-in`, the run type) is a **pill**.

Circular dependency - A chain of namespaces that leads back to its own start (A → B → A). The C# compiler allows it, so it is a modularity smell, not an error. `detect_circular_dependencies` finds them.

`Compression Rules` - A named **rule set** (not a single rule) that decides how strongly code blocks and sections are shortened when the token budget is tight: scoring multipliers by accessibility, role and path, a tiebreaker order, and the trimming mode `Greedy` or `Sweep`. One entry is therefore a set of rules, not "a compression rule". Page `Compression Rules`; a context template can reference its own rule set. The output section `COMPRESSION_LEGEND` is always rendered and explains the notation the rules produce.

Constellation - An exportable and importable **bundle of master data**: several library entries in one JSON file, validated against a schema. Managed on the `Constellations` settings page; `aicb import --file <file.json> --mode SkipExisting|Replace` applies one (add `--preview` for a dry run).

`Context Builder` - The working area for one solution: the `Solution Tree`, the six configuration panels (`Prompt`, `Sections`, `Graph Selection`, `Selection Engine`, `Test Description`, `Export Output`) and the generated document. One tab per solution.

Context document - The output of an export: a dense Markdown document in the `AI-Builder-MD` format, written to a `.md` file and meant to be handed to a language model. The notation inside is either tag or YAML (see `Format`).

Context template - The configuration root of an export: which prompt and which MD profile it uses, which detail preset and expansion strategy apply at each of the four detail levels, which test description, compression rules, pipeline profile and quality profile are attached, and its own switches (line numbers, quality metrics, config files that may contain secrets, layout legend). The settings page is `Templates`. Not to be confused with a run template (`Run Templates`).

12.3 D - G

Dead code - A symbol that nothing in the **analyzed scope** references. Note: reflection, dependency injection resolved by string, XAML and external callers are invisible to the analysis, so "no callers" is a lead to verify, not proof. `find_dead_code` lists suspects.

Detail level - The four-step axis that says **how much** of a symbol enters the document: `Compact`, `Normal`, `Detailed`, `Source`. It is resolved per node in the `Solution Tree`; `Source` behaves as `Detailed` plus the source-code block.

Detail preset - A master entry that describes **how** the content of a detail level is written. A context template has one slot per level. Rule of thumb: the level says how much, the preset says how. Page `Detail Presets`; the slots are under `Selection engine (per detail level)` in the template editor.

Drift - A measured divergence between two states that should match. `server_info` reports `CONFIG DRIFT` (the active MCP profile changed after the server started), `VERSION DRIFT` (the running binary is older than the database it reads) and `ANALYZER DRIFT` (the analysis code moved ahead of the build). `check_solution_config_drift` checks whether a solution's configuration still fits its code. Source staleness is a different thing; see **Staleness**.

Edge - A modeled relationship between two symbols: calls, is called by, depends on, implements, inherits from, XAML names a type, XAML binds a member.

`Exclude Namespaces` - A named list of namespaces that the analysis skips. Note the label is identical in both places it appears: `Settings > Exclude Namespaces` is the library editor, `Workspace > Profiles > Exclude Namespaces` is the per-solution picker.

Exit code - The value the command line returns. Seven values with a fixed meaning:

Code	Meaning
0	Success
1	User error (invalid arguments, missing files)
2	Runtime error (unexpected exception)
3	Cancelled (`Ctrl+C`)
4	Database migration failed
5	MSBuild not found (`aicb analyze` only)
6	Quality gate failed (`aicb analyze --fail-on` only; the document is still produced)

Expansion strategy - Controls **how far** the selection reaches from a marked node into the graph: a strategy mode (`Reachable`, `Select`, `Frontier`), a method depth and a type depth, eight switches (callers, callees, used-by members, inheritance hierarchy, interface implementations, created types, used types, injected dependencies) and four stop conditions (exclude framework types, exclude external assemblies, exclude test classes, exclude generated code), plus an optional total node limit. A context template has one slot per detail level for the GUI tree, plus `Expansion around the MCP start point` for the MCP server and the CLI. Not to be confused with compression: expansion decides **what gets in**, compression how briefly it is then written.

Export - The act of writing the context document, and the document itself. Other kinds of output carry their object: master-data export, session export, sidecar export, usage-report export.

Fact - A measured property of a symbol that the analysis derives from the code: cyclomatic complexity, lines of code, called methods, used types, side effects. A fact is measured; a semantic axis is resolved and carries a provenance.

Facet - The one task axis of the MCP. Nine values: `general`, `exploration`, `refactoring`, `debugging`, `review`, `testing`, `documentation`, `architecture`, `performance`. A facet has three effects: it picks a context template, it composes a tool menu, and it contributes a guiding sentence. You address it per call with the `facet` parameter of `prepare_task`, `pack_for_task` and `export_markdown` - a call parameter, not a server state.

False positive - A finding that the analysis reports and that is not one. Each producer has documented false-positive classes; where a producer knows one, the finding names it. Treat a finding as a lead to verify.

Fan-in - The set of places that name a symbol: "who calls this?". `find_usages` lists the direct references (one level); `impact_of_change` adds the transitive closure. The output section `METHOD_USED_BY_GRAPH` is the rendered fan-in of a method.

`File-Set-Hash` - A checksum over the set of a solution's source files. If it has not changed, an existing snapshot can be reused instead of re-analyzing the solution; a plain `refresh_session` compares it before it re-runs the analysis.

`Findings` - The findings parsed out of the **language model's answer**, shown in the `Findings` region of the `Reasoning` panel. aicb's own quality findings are `Insights` - the product's tab is named `Insights` for exactly this reason. In this manual, `Findings` means only the model-derived kind.

Format - The inner notation of the context document: `Tag` (the established AI-Builder tag Markdown) or `YAML` (idiomatic YAML, the product default). It is a notation of the same content, not a different selection, and the file keeps its `.md` name either way.

12.4 H - L

Handshake - The text the MCP server hands a client when it connects: what the server can do, how to use it, which conventions apply. It is composed of blocks, and a block whose tools the active profile does not expose is left out.

Hook - A script that your agent environment runs at a defined point. `aicb init --hooks` installs the aicb symbol guard as a hook; `install_agent_hooks` does the same for a project that is already set up.

Impact - The **transitive** fan-in closure: everything that could be affected by a change to a symbol, including consumers a compiler does not check. `impact_of_change` reports it with a `risk` level. Note: the risk is measured on the symbol's fan-in, not on the semantics of your specific edit - a `high` rating on a purely additive change is expected and is not a stop sign. `find_usages` lists only the direct references.

Insight - A finding of the built-in code-quality analysis: a place where aicb found something worth mentioning, with a severity, a category and one or more sites. Shown in the `Insights` tab and returned by `list_insights` / `get_insight`; also rendered into the output section `QUALITY_FINDINGS`.

`inPool` / `outOfPool` / `uncatalogued` - The pool states of MCP tools. The **pool** is what the active profile exposes right now. `inPool.core` are the tools in every profile, `inPool.extras` the rest of the exposed set, `outOfPool` tools that exist but are not exposed (a name there proves the tool is real), and `uncatalogued` tools that no menu and no bundle names and that only `AICB_MCP_TOOLS` can expose. `list_skills` reports the current split; the server registers 82 tools in total.

Layer - An architecture layer such as `Domain`, `Application` or `Infrastructure`.

Layer profile - A named rule set that maps types to layers and sets how strictly a violation is judged: `Advisory` (violations are warnings; the default for new profiles) or `Strict` (violations are critical). Pages: `Settings > Layer Profiles` is the library, `Workspace > Profiles > Layer Profile` is the picker for one solution.

Library - The collection of master entries (the library pages under `Settings`). Not an assembly and not a NuGet package.

12.5 M - N

`Manual` - Four different things carry the word: the run type `Manual` (one call per trigger), the snapshot type `Manual` (created and named by you), a manual override on a tree node, and the provenance `Manual` of a value you typed yourself.

Master data - The editable library entries an export is assembled from. Every master entry has the same shape: an id, a name, a description, built-in/overridden/hidden flags, an active and a default marker, and a reference count. The settings pages are the library editor; the matching pickers under `Workspace > Profiles` only choose, they do not create.

MCP - Model Context Protocol, the protocol over which an AI agent calls the tools of an external server. `aicb mcp` starts that server.

MCP profile - A named combination of: which tools a client sees, which facets are active, how the output is shaped, how the session behaves, and which handshake text goes along. Its own `MCP Profiles` page is in the main sidebar group `MCP`, with the sub-tabs `Facets`, `Tools`, `Output`, `Session` and `Instructions`; it is not a Settings page. Four profiles ship with the product (`Default`, `Full Select`, `Refactoring Focus`, `Debugging Focus`). Pin one for a server process with `aicb mcp --mcp-profile <id>`; `list_mcp_profiles` lists the ids.

MCP session - An analysis result held in the server process, addressed by a `session_id`, with no name and no database row; it is gone when the process ends. Every tool that takes a session also accepts the absolute `.sln` path directly and analyzes on first use - that is `Self-Init`, and it makes `analyze_solution` optional. The cache is keyed by `.sln` path within one server process, with a sliding expiry (default 90 minutes, at most 8 sessions); a second process analyzes from scratch. `AICB_MCP_SESSION` tunes `ttlMinutes`, `maxSessions` and `sweepMinutes`.

MD profile - A container with one slot per output-section type (27 types, six of them graph types; four are staged slots for classes, methods, interfaces and enums), and in each slot a tag schema. The navigation entry is literally `MD Profile` (singular), even though the page manages several profiles. The MD profile is the assignment, the tag schema is the content.

`measure` - An MCP tool that reports how large a read-only query's answer would be, in tokens, without returning the answer. It still runs the query, so it saves context, not time.

Member - A part of a type: method, property, field, event, constructor.

Migration - A numbered step that advances the database schema. Missing steps run at the first start after an update, so a database is migrated when you first open it with a newer version. Note: this is one-way - an older build cannot read a database that a newer build has already migrated. If you run two versions side by side, give one of them its own database path under `Settings > Storage`.

Mode - Two kinds are common: the layout mode (the five chips in the Context Builder) and the auto-refresh mode (`Off`, `Reactive`, `Proactive`).

Model profile - A named LLM configuration: provider and kind, endpoint, model name, token limits, temperature, context window and prices per million tokens. The `Test Connection` button checks the profile against the real endpoint. Page `Model Profiles`.

Namespace - A C# namespace. In the `Solution Tree` it sits between the project and its types.

Navigation - The left sidebar of the main window. It has four groups: `Quick Access` (`Start`), `Workspace` (`Context Builder`, `Workspace`, `Run Templates`), `Configuration` (`Templates`, `Settings`) and `MCP` (`MCP Profiles`, `MCP Usage`). Clicking an entry opens it as a tab.

Node - One entry of the `Solution Tree`: solution, project, namespace, type or member.

12.6 O - R

Omission - The handshake rule that leaves out a block whose tools the active profile does not expose. Do not use the word for the separate case in which a `batch` result is too large to fit: that one is reported as omitted with a pointer to the direct call.

Overlay - A modal layer over the work area: the loading overlay while a solution is analyzed, and the export overlay while a document is rendered. The window stays responsive behind it.

Overridden - A built-in master entry that you have edited. The original is kept and `Restore Built-In` brings it back. Both `Built-in` and `Overridden` appear as pills on the card.

Page - A whole surface with its own header, reached from the sidebar (`Layer Profiles`, `General`, `MCP Usage`). A register inside a page is a **sub-tab**; a bounded block inside a page with its own heading is a **panel**.

Panel - A delimited block inside a page or tab with its own heading, for example `Prompt`, `Sections` or `Graph Selection`.

Pill - A non-clickable marker on a card or row: `Built-in`, `Overridden`, the run type. A clickable one is a **chip**.

Pipeline profile - Token budget and trimming behavior of the export, plus two switches for snapshot tabs. `BudgetedTrimmingEnabled` is off by default; `MaxTokenBudget` defaults to 60,000 tokens (floor 8,000) and `MaxOvershootPercent` to 50 (clamped to 0-1000). Page `Pipeline Profiles`. Note: do not confuse it with the run type `Pipeline` (not yet released).

Pool - The set of MCP tools the active profile delivers right now. See `inPool` / `outOfPool` / `uncatalogued`.

Profile - Seven different things carry the name:

Label	What it controls	Where
`Layer Profiles`	type → architecture layer, plus strictness	`Settings` library + per-solution picker
`MD Profile`	one slot per output-section type, with the chosen tag schema	`Settings`
`Quality Profiles`	which insight producers run	`Settings`
`Test Profiles`	what counts as a test project and as a test method	`Settings` library + per-solution picker
`Pipeline Profiles`	token budget and trimming behavior of the export	`Settings`
`Model Profiles`	LLM endpoint, token limits, prices	`Settings`
`MCP Profiles`	which MCP tools a client sees, plus session behavior	`Settings`

Project - A `.csproj` inside the solution. A project that targets several frameworks appears in the analysis once per framework.

Prompt - Three related things. (a) The master entry `Prompt` (a prompt template) that a context template references. (b) The five fields of the `Prompt` panel in the Context Builder - `System Role`, `Instructions`, `Goal / Task (User Prompt)`, `Constraints / Rules`, `Additional Context` - from which the text actually sent is assembled; the first two are initialized from the referenced prompt template, the other three are per-session. (c) The assembled text itself, which is what goes to the model.

Provenance - The recorded origin of a resolved semantic value, printed in the output document so you can tell a measured value from a guess: `fact` (a deterministic Roslyn fact), `ai` (asserted by the developer in a `/// <ai>` comment), `verified-absent` (the developer declared "none"), or `inferred` (a heuristic). A semantic axis without provenance is unknown, not empty.

Quality gate - The option that makes `aicb analyze` fail when findings at or above a severity are found: `--fail-on "<expression>"` (for example `critical>0 OR ce-max>50`). The document is still produced; the exit code is 6. An empty expression is rejected rather than passing silently. This is the option that makes aicb useful in a CI pipeline.

Quality profile - Switches the insight producers on and off - it decides which kinds of finding are produced at all, together with their thresholds, whether findings are dismissable, and which action mode the `Apply` button uses (`DirectApply` or `ConfirmDialog`). Not to be confused with severity: the profile decides **whether** a check runs, the severity how heavy its result weighs.

Reasoning - The panel that shows what happened in a run: what the model answered, what it thought, which tools it called, which `Findings` were parsed out of the answer, and the evaluation.

Run - A pass in which aicb sends an assembled context to a language model and collects the answer. Labels such as `Run Templates`, `Active Run` and the layout mode `Run` keep the word. Not to be confused with the traversal depth of the tree expansion.

Run template - The configuration of a run: which run type, which context template, a description, and an optional suggested text for the goal field of a manual run. Page `Run Templates`. A run template **selects** a context template; it is not one.

Run type - What a run does per trigger. The released type is `Manual` (one model call per trigger, or a pure Markdown render). Two further types are **not yet released**: `Iteration` (one call per tree node of the selection) and `Preselection` (the model selects first, then the actual processing runs in two or three stages). A fourth value, `Pipeline`, exists in the data format as a placeholder for a planned multi-step run with optional loops and splits; it is **not yet released** and cannot be selected.

12.7 S

Section - A named part of the generated context document, for example `SPEC`, `META`, `QUALITY_FINDINGS` or `LAYER_MAP`. A labeled group on a settings page is a *settings section*, not a document section.

Self-Init - The behavior that makes `analyze_solution` optional: every MCP tool that takes a session also accepts the absolute `.sln` path instead and analyzes the solution on first use. The returned `session_id` is reused for all further, cheap calls.

Semantic axis - A derived value on a type or method - role, layer, domain, responsibility, side effects and others - resolved from several sources. The provenance travels with it in the output document, so you can see whether a value was measured, asserted or inferred.

Session - A named, saved working state of a solution that you can reopen and continue: the tree selection, the detail-level overrides per node, the prompt text, and the template as it was when you saved. `Save Session` (or `Ctrl+S` in the Context Builder) always opens the name dialog, prefilled for an existing session. Auto-Save and the non-interactive save-before-close path update it without asking. The `Sessions` sub-tab of the `Workspace` page lists them. See "Terms that are easy to confuse" for the difference from an MCP session.

`session_id` - The handle of an MCP session. You normally do not have to manage it: pass the `.sln` path and let `Self-Init` do the rest.

Sidecar - The git-tracked configuration file next to a solution: see `.aicb.json`.

Side effect - What a unit touches outside its own memory. The vocabulary is closed and has eight values: `io`, `network`, `database`, `serialization`, `logging`, `cache`, `messaging`, `unknown`. You meet them in `find_by_side_effects(effect: ...)`, in the output section `SEMANTICS` and in the quality-profile checkbox `Side-effect concentration (methods mixing 3+ effect categories)`. Worth remembering: the classification judges by **contact, not by topic** - `Path` and `MemoryStream` carry no `io` effect, while `File` and `FileStream` do. A cache a type holds in memory is not an outside-world effect.

Skill - An instruction file that an agent loads. `aicb init --skills=all` writes the aicb context skill plus the review pair and the usage check; the default `context` writes only the context skill.

Snapshot - A frozen analysis of a solution at a point in time. Runs are pinned to a snapshot so that a later re-analysis cannot change a finished run. A snapshot is created automatically on every fresh analysis (type `Auto`) or by you via `Create Manual Snapshot` (type `Manual`). `Max snapshots per solution` limits how many automatic snapshots are kept; snapshots you saved yourself are never deleted and do not count towards the limit. `Diff Two Snapshots` compares two of them. The `Snapshots` sub-tab of the `MCP Usage` page shows something else - see "Terms that are easy to confuse".

Solution - A Visual Studio solution: the `.sln` file plus everything it contains. It is the unit aicb analyzes; without a solution nothing happens. You meet it in the `Solutions` list, the `Solution Tree`, the CLI option `--sln` and the MCP parameter `sessionId`, which accepts an absolute `.sln` path. Note: a solution entry in the database is the stored record with its active profile assignments - say "the solution entry" when you mean that.

`Solution Tree` - The tree on the left of the Context Builder: solution → project → folder/file → type → member. A namespace is not a tree level. Here you choose what goes into the document.

`Solutions` - The list on the left of the `Workspace` page. The word appears only there.

Staleness - The state in which the analysis graph is older than the source files, because something was edited since the analysis. A session-bound answer from a stale session carries a `staleness` trailer (a JSON member or Markdown comment); a current answer carries none. Note the difference from **incompleteness** (`incompleteProjects`): staleness means the source moved on, incompleteness means the project references could not be resolved because the solution was not built. A `not_found` on a type you just wrote is staleness and is cured by an ordinary refresh; a plausible but too short fan-in that discloses `incompleteProjects` needs `refresh_session(force: true)` after a restore or build.

Style - A cross-cutting working guideline that adds text to the server instructions without changing the tool set. The product ships two: async/concurrency correctness and security review. A style that belongs to one kind of work sits on that facet instead.

Sub-tab - A register **inside** a page or surface: `Facets` in the MCP profile editor, `Details` in the Context Builder. The strip at the top of the window is the tab strip; a sidebar entry opens a page as a tab.

Suppression - A single insight that you deliberately hid. In the `Insights` tab you mark a line as intentional; that action stores the suppression in the local database. It reaches the committed sidecar's `suppressions` array only when you explicitly run `Export Config`; the reading path combines both sources. The reading surfaces (`Insights`, `list_insights`, `get_insight`) honor it; the quality gate and `solution_metrics` keep counting it. Do not confuse it with dismissing a finding as seen: a dismissal is a local database record, while an exported suppression can become a shared decision.

Symbol - A C# type or member. A glyph on screen is an `Icon`.

12.8 T - Z

Tab - An entry in the top strip of the window, and the surface it shows. Closing one with `Ctrl+W` closes the innermost first: an open file before its solution tab, a solution tab before the page tab.

Tab strip - The row of tabs at the top of the window.

Tag schema - The actual render configuration of one output-section type: which marks are used, in which order, in which layout. Per section type there are several built-in schemata and optionally your own; exactly one is the default. Page `Tag Schemata`; the `Sections` panel picks one per output section for a run. The MD profile is the assignment, the tag schema is the content.

`Templates` - The settings page that manages context templates (see `Context template`). Do not confuse it with `Run Templates`, which manages runs.

Test profile - Makes test detection pinnable: which projects count as test projects (`TestProjectRules`, matching the project name with `Contains`, `StartsWith`, `EndsWith` or `Exact`) and which method attributes mark a test case (`TestAttributeNames`, matched as a substring). Built-in presets are `default` (project name contains `Test`, `Tests`, `Spec` or `Specs`; attributes `Fact`, `Theory`, `Test`, `TestMethod`, `TestCase`), `xunit`, `nunit` and `mstest`. Pages: `Settings > Test Profiles` is the library, `Workspace > Profiles > Test Profile` is the per-solution picker.

Token budget - The upper limit on how large the generated document may become. When it is reached, compression and trimming take effect. You meet it on the `Max MD size` slider in the Context Builder header, in the template editor as `Token budget (MCP render)`, in the pipeline profile, and in the confirmation before a run. The floor is 8,000 tokens; the global default is 60,000. Precedence: explicit call parameter > MCP profile > template > global pipeline profile.

Tool - A single callable function of the MCP server, for example `find_usages`. The word also appears in the protocol concepts `tools/list` and `tools/call`.

Type - A class, struct, interface, record or enum. The third level of the `Solution Tree`.

Unit - Everything with an **executable body**, and therefore everything the analysis can judge: not only declared methods, but every property, indexer and event accessor with a body, every constructor including the member initializers it runs, top-level statements and Razor components. An auto-property accessor (`get;`) has no body and is not a unit. You meet the word in MCP answers as a counter (`methodsScanned`, "over N units") and in hits such as `Type.Name.get`. The difference from "method" matters: a private method that is called only from a setter would look uncalled if accessors were not units of their own.

Usage recording - The local record of which MCP tool was called how often, how fast and with which outcome. It is visible and deletable on the `MCP Usage` page (six sub-tabs: `Tools`, `Calls`, `Notes`, `Errors`, `Clients & protocol`, `Snapshots`) and through `usage_report`. Nothing of it leaves your machine - it stays in the local database. With

`AICB_MCP_ERROR_TEXT=off` you reduce the one free-text error field to the exception type. There is no switch to turn the recording off; you can clear the data at any time.

Usage snapshot - An imported usage report kept beside your own live numbers on the `MCP Usage` page. Imported reports are never merged into the live data; you can compare one against the current numbers and delete it again. The export writes the same JSON that `usage_report` returns, so a report can travel between machines.

Workspace - The page that manages all known solutions. On the left is the `Solutions` list; on the right are four sub-tabs: `Overview`, `Profiles`, `Sessions` and `Snapshots`. Note that `Workspace` is also the name of the sidebar group that contains this page, so say "the `Workspace` page" when the distinction matters.

12.9 Terms that are easy to confuse

Four pairs cause most of the confusion in this product. Read these first; the table after them covers the rest.

`Insights` **vs** `Findings` - `Insights` are aicb's own quality findings: the `Insights` tab, `list_insights` and `get_insight`, and the output section `QUALITY_FINDINGS`. `Findings` are the findings parsed out of the language model's answer, shown in the `Findings` region of the `Reasoning` panel. In this manual, `Findings` always means the model-derived kind.

`Session` **vs MCP session** - A `Session` is the named, saved working state of a solution that you reopen and continue. An **MCP session** is an in-memory analysis result inside the server process, addressed by a `session_id`, with no name and no database row. This manual writes `MCP session` whenever both could be meant. There is a third place the word appears: the `Session` sub-tab of the MCP profile editor, which configures auto-refresh (`Off`, `Reactive` or `Proactive`) and its scope, not a session. The headless session cache is configured through `aicb.mcp.json` or `AICB_MCP_SESSION`, not in that GUI tab.

`Snapshot` **vs usage snapshot** - A `Snapshot` is a frozen analysis of a solution at a point in time; runs are pinned to one. The `Snapshots` sub-tab on the `MCP Usage` page holds imported usage reports instead. Call the second one a **usage snapshot**.

`Workspace` **vs** `Solutions` - `Workspace` is the sidebar group and the page inside it; the page title is `Workspace`. `Solutions` is the list on the left of that page and the only place the word appears. This manual writes "the `Workspace` page" and "the `Solutions` list".

Term alone	Means	The other meanings are called
`Session`	the saved working state in the GUI	`MCP session`; the `Session` sub-tab of the MCP profile editor
`Snapshot`	the frozen analysis of a solution	usage snapshot
`Findings`	findings parsed from the model's answer	`Insights` (aicb's own)
`Template`	- ambiguous	context template, run template
`Profile`	- ambiguous	layer profile, MD profile, quality profile, test profile, pipeline profile, model profile, MCP profile
`Mode`	- ambiguous	layout mode, auto-refresh mode
`Axis`	the solution configuration (layer, exclusions, test)	detail level, severity, facet

Term alone	Means	The other meanings are called
`Card`	a list entry	sub-tab (the register of an editor)
`Section`	a named part of the context document	a settings section (a labeled group on a settings page)
`Symbol`	a C# type or member	`Icon` (a glyph on screen)
`Library`	the master-data collection	assembly, NuGet package
`Manual`	- ambiguous	run type `Manual`, snapshot type `Manual`, manual override, provenance `Manual`
`Run`	a pass that sends context to a language model	traversal depth of the tree expansion
`Level`	the detail level	the font size under `Settings > General`
`Export`	the context document	master-data export, session export, sidecar export, usage-report export
`Block`	a handshake block	the `staleness` trailer of an answer
`Drift`	- ambiguous	`CONFIG DRIFT`, `VERSION DRIFT`, `ANALYZER DRIFT`; staleness
`Slot`	a rendering or template slot	the `slot` parameter alias (an older name for `facet`)
`Bundle`	the opt-in tool group (`api-surface`)	the rendered context document
`Dispatch`	running several queries in one `batch` or `measure` round trip	interface dispatch (a C# call through an interface)
`Omission`	the handshake rule for unexposed tools	an oversized `batch` result omitted with a pointer

Four file names look alike and mean four different things:

File	What it is	Where it lives
`aicb.acb`	the application database (SQLite)	where `Settings > Storage` points, default `{BasePath}\user-data\aicb.acb`
`.aicb.json`	the **sidecar** of a solution (layer, exclusions, test detection, suppressions)	next to the `.sln`, committed to version control
`aicb.mcp.json`	the **server** configuration of the MCP for headless operation	with the server data
`.mcp.json`	the **client** configuration of your agent (what it starts)	in your project; `aicb init` adds the `aicb` entry

Finally, three product terms that look like each other but are not:

- `Layer Profiles` (plural, the library under `Settings`) against `Layer Profile` (singular, the picker for one solution). The same singular/plural rule separates `Test Profiles` from `Test Profile`. `Exclude Namespaces` is spelled the same in both places, so use the location to distinguish them: `Settings > Exclude Namespaces` against `Workspace > Profiles > Exclude Namespaces`.

- `Run Templates` (the configuration of a run) against `Templates` (the context templates). A run template selects a context template.
- `Pipeline Profiles` (the token budget and trimming behavior of an export) against the run type `Pipeline`, which is not yet released.