CODESYS File-Based Storage 1.0.0.0

CONTENT

Last Modification Date: 08/13/2026

1 CODESYS File-Based Storage: Technical Whitepaper

This whitepaper is written for CODESYS users who are also developers: people who work with Git, run CI/CD pipelines, review code, and want to understand exactly what CODESYS 3 FBS does, what it stores, and how far its openness reaches. It describes the current, shipping behaviour.

1.1 Terminology used in this whitepaper

This document deals with two products and three storage formats, which are easy to confuse. Each has a fixed name, used consistently from here on:

- **CODESYS 3 Binary**: the classic storage format of CODESYS 3: one monolithic binary archive per project (`.project` , `.library` , `.projectarchive`).
- **CODESYS 3 FBS**: File-Based Storage as it ships today: the folder-based storage format for CODESYS 3 projects, and the subject of this whitepaper.
- **CODESYS 4 projects**: Projects of the next CODESYS generation, a separate product, which stores its projects file-based natively.
- **the file-based text format**: the on-disk text format itself, common to CODESYS 3 FBS and CODESYS 4. It originates with CODESYS 4; CODESYS 3 FBS reads and writes the same format. That is what "shared format" means throughout this document.

1.2 1. What File-Based Storage is

CODESYS 3 FBS stores a CODESYS 3 project as a **folder tree on disk** instead of a single CODESYS 3 binary archive (`.project` or `.library`). Every folder and programming object becomes a separate file or directory. The layout on disk mirrors the project tree you see in CODESYS.

Why CODESYS 3 FBS exists is best read in order of priority:

- **CODESYS 3 FBS is primarily the conversion tool towards CODESYS 4**. CODESYS 4, the next generation of CODESYS, stores its projects file-based natively. That is where the file-based text format comes from. CODESYS 3 FBS brings that same storage model back to CODESYS 3: it writes today's projects in the same format CODESYS 4 uses, so converting a CODESYS 3 Binary project into CODESYS 3 FBS is the first step of the move to CODESYS 4. Section 2 describes this path.
- **CODESYS 3 FBS is also the new foundation for CODESYS Git 2.0.** Because the CODESYS 4 storage model now also exists on CODESYS 3, version control gains a new basis. CODESYS Git 2.0 builds on it and combines Git's branching-and-merging model with CODESYS's knowledge and consistency rules of the project; it is the recommended version-control mechanism for CODESYS projects (section 3).
- **CODESYS 3 FBS is open to external tooling by design, even if that openness is not its primary purpose.** The storage is text, JSON and XML, so tools outside CODESYS (a plain `git` client, `grep` , a web-based review platform) can read/write it, and within documented limits that is supported. CODESYS 3 FBS was not developed to make external tooling a primary workflow, though: projects are edited in CODESYS, and versioned with CODESYS Git 2.0 (section 9).

Three principles define the format:

1. **Mostly one object, one file.** The on-disk structure mirrors the project tree. There are a few deliberate exceptions: an object with children becomes a directory, and some objects bundle related parts into a single file; a property, for example, keeps its get/set accessors together (see section 4).
2. **Text where possible.** ST source is plain text; project metadata is JSON. Only content that has no clean textual form falls back to XML (see section 6).
3. **Stable identity.** Objects and folders carry a stable GUID, so renames and moves do not break cross-references and do not look like delete-plus-create in a diff.

CODESYS 3 FBS is implemented as a CODESYS 3 `IProjectFormat`: from the IDE's point of view an `.fbsproj` folder is "just another project format" it can open, edit, and save. Conversion from a CODESYS 3 Binary project into the folder format is an explicit action.

1.3 2. The road to CODESYS 4

The primary purpose of CODESYS 3 FBS is to carry existing CODESYS 3 Binary projects toward CODESYS 4. To see how it does that, it helps to lay out the concrete formats in play, how they relate, and which direction data can flow.

1.3.1 2.1 The formats in play

Three families of formats are involved: the CODESYS 3 Binary formats that conversion starts from, the CODESYS 3 FBS folder formats it produces, and the CODESYS 4 project formats it targets. Each is introduced briefly here so that the conversion paths in the next section are unambiguous; the full on-disk detail is in section 4.

CODESYS 3 Binary formats

- `.project`: a device project as a single monolithic binary archive. The classic format, and opaque to version control.
- `.library`: a library in the same binary form.
- `.projectarchive`: A container carrying a library or project but also containing all repositories and settings.

CODESYS 3 FBS formats

- `.fbsproj`: a device project, stored as a folder tree. The file-based counterpart of `.project`.
- `.fbslib`: a library, stored as a folder tree. The file-based counterpart of `.library`, and the one format shared outright with CODESYS 4 projects.
- `.fbsws`: a workspace, stored as a small JSON file listing member projects by relative path and used to open them together. No CODESYS 3 Binary counterpart.
- `.fbslibpack`: a library-package archive, packed into a single file for distribution. No CODESYS 3 Binary counterpart.

In every CODESYS 3 FBS format, each object carries a stable `ObjectGuid` and a set of object properties.

CODESYS 4 project formats

- `.fbslib`: the same library format CODESYS 3 FBS writes. A shared format: one library folder for both products.
- `.fbsdev`: the CODESYS 4 device project, the counterpart of `.fbsproj` and `.project`. Reaching it requires one additional explicit conversion step (section 2.2).

CODESYS 4 has no concept of object GUIDs or object properties; section 2.3 explains why one format nevertheless serves both products.

1.3.2 2.2 The conversion tool

The entry point is the explicit CODESYS 3 Binary → CODESYS 3 FBS conversion. It is a deliberate, visible action (a project never changes format silently) and it is available in two places:

- as an IDE command, converting a CODESYS 3 Binary `.project` / `.library` into an `.fbsproj` / `.fbslib` folder;

- via the Python scripting bridge (`project.convert_to_fbs(path, …)` , see section 8), for converting in bulk or inside a pipeline.

What the conversion buys differs by project type:

- Libraries. CODESYS 3 FBS and CODESYS 4 projects share the `.fbslib` library format outright: a library converted to CODESYS 3 FBS is already stored in the format that CODESYS 4 reads and writes.
- Device projects. For `.fbsproj` projects, the mapping to the CODESYS 4 `*.fbsdev` format goes through an additional explicit conversion step, which will become available in upcoming CODESYS 3 FBS versions.

1.3.3 2.3 One format, optional identity: why CODESYS 3 FBS and CODESYS 4 projects can share it

The central design decision is that CODESYS 3 FBS and CODESYS 4 projects use the same on-disk file-based text format. They differ in one respect: the stable `ObjectGuid` and the object properties exist only in CODESYS 3 FBS, and they are carried optionally.

Concretely:

- A CODESYS 3 FBS writer emits the GUID and properties (inside the `(* METADATA *)` comment for source files, or the JSON meta for other files), but omits them entirely when empty, rather than writing empty placeholders.
- A CODESYS 3 FBS reader reuses a GUID that is present and *generates a fresh one when it is absent*, which is what a file written by CODESYS 4 looks like.
- CODESYS 4 neither produces nor consumes GUIDs and properties; it simply ignores metadata it does not need.

The same repository of text files is therefore meaningful to both products: a file authored in one is loadable in the other, and identity degrades gracefully instead of breaking.

1.3.4 2.4 Features that exist only in CODESYS 4

The shared format is a superset: CODESYS 4 can express constructs that CODESYS 3 has no concept of. The relevant behaviour is what happens when CODESYS 3 opens a project that uses one of them: it does not fail, but reports the issue and opens the affected project read-only, so a CODESYS 3 session can never overwrite a CODESYS 4-authored file with a lossy CODESYS 3 interpretation.

Three such CODESYS-4-only features exist today, both in the Library Manager:

- **Relative library references.** CODESYS 4 can reference a library by a relative path. CODESYS 3 has no relative-reference concept: on load it reports an error ("relative libraries are not supported"), keeps the reference visible as an unresolvable entry, and sets the project read-only.
- **Semantic version numbers for placeholders and references.** CODESYS 3 library versions are numeric with 2 to 4 components, optionally a trailing wildcard (`3.5.*`) or the same-as-container marker (`=`). CODESYS 4 can use semantic versions, which are not valid CODESYS 3 versions. On load, CODESYS 3 first tries to resolve the intended version through `libraries.lock.json` ; if that succeeds it uses the locked version (reported as information); if not, it reports an error, substitutes a placeholder version, and sets the project read-only.

The third is not about the content of a project but about how projects are grouped:

- **The workspace as a container for several projects.** In CODESYS 4 a workspace is a real container: the projects it lists are opened and worked on together. CODESYS 3 has no such concept, it has exactly one project open at a time. It therefore reads a `.fbsws` file as a *selector* rather than a container and resolves it to a single project before opening it: if the file lists exactly one project CODESYS V3 can open, it opens silently. Otherwise CODESYS 3 asks which one to open (in scripting the member is picked by index, see section 8). The other entries are alternatives for the next open, not projects loaded alongside. They stay in the file untouched. This is the one CODESYS-4-only feature that does **not** make the project read-only, the project that is opened is a perfectly ordinary FBS project.

1.4 3. Version control: CODESYS Git 2.0 on CODESYS 3 FBS

Because the CODESYS 4 storage model now exists on CODESYS 3, version control gains a new foundation. CODESYS Git 2.0, the CODESYS Git integration built on that foundation, is the recommended version-control mechanism for CODESYS 3 FBS projects.

1.4.1 3.1 What CODESYS 3 FBS provides for version control

A monolithic binary archive is a black box for any version control system: it cannot be diffed or merged, and a one-line change to one function block rewrites the whole file. With FBS, every object is its own text file. That changes two different things, and it is worth keeping them apart.

Inside CODESYS 3, version control finally works on the project itself. Real per-object diffs, merges that conflict only when both sides touched the *same* object, feature branches, blame and history down to a single function block, cherry-picks and reverts. This is where FBS pays off, and it is where Git work belongs: driven from the IDE by the CODESYS Git integration, which understands the format and the project model.

Outside CODESYS, the project becomes readable. Because the source is text, a repository platform shows a CODESYS project the way it shows any other source code. That makes a few workflows possible with no IDE installation at all:

- **Reviewing a merge request in GitLab** (or a comparable platform): the changed Structured Text source is right there in the web diff, so reviewers can read and comment on it in the browser.
- **Reviewing Structured Text code**: declarations and implementations are ordinary text, so ordinary code review applies.
- **Sizing up a revision at a glance**: which objects a commit touched, and how much changed. Enough to judge the scope of a change without opening the project.

These three workflows are *review*. It is not an invitation to run a native Git workflow on the folder: we do not recommend performing Git operations, such as staging, committing, branching, merging, resolving conflicts, with a plain Git client directly on an FBS project. Two reasons:

- **Consistency rules.** A CODESYS project is an object model with cross-object references and metadata that has to stay in step. A merge that is clean at the text level is not automatically a project that loads. Keeping commits and checkouts coherent is exactly what the CODESYS Git integration does (section 3.2).
- **Graphical editors need their comparers.** LD, FBD, SFC and CFC are compared and merged in the CODESYS comparison view, which shows the network itself rather than the text behind it. On disk they are XML, or a new text format (section 5.3) — a raw text diff of such an object may be hard to read at best and misleading at worst.

1.4.2 3.2 Why CODESYS Git 2.0 is the recommended mechanism

Plain Git commands work on the folder tree; the format is designed so they can. But only the CODESYS Git integration, version 2.0.0.0 or newer, understands the file-based text format, and that gives you two guarantees that raw Git cannot:

- **Everything the project needs is committed.** The integration ensures that all files that belong to a project (every object file, its metadata, and the supporting structure) are staged and committed together, so a commit is never missing a piece that the project needs to load and build.
- **The committed state stays consistent.** Commits are made against a validated snapshot of the project rather than a half-written working tree, so a checkout always reconstructs a project that CODESYS can open cleanly: no dangling references, no partially-written objects, no mixed-state commits.

Use CODESYS 3 FBS together with CODESYS Git 2.0 whenever it is available. Working on the folder with a raw Git client is possible, but it is only a fallback; the ground rules for it are in section 9.

1.4.3 3.3 Merge-conflict awareness

Because each object is its own text file, most merges resolve cleanly. When a merge *does* conflict, the version control system writes its usual conflict markers (`<<<<<<<` , `|||||||` , `=======` , `>>>>>>>`) into the affected file. CODESYS 3 FBS detects these markers on load and save: as long as a file still contains conflict markers, CODESYS reports the conflict and refuses to load or save that object, rather than silently corrupting it.

1.4.4 3.4 What belongs in version control, and what does not

CODESYS 3 FBS keeps some folders that are derived caches and must be excluded from source control:

- `.auxiliaries` : a cache of data CODESYS 3 FBS (or an add-on) can regenerate from the committed project (indexes, pre-processed artifacts). The project must load and build correctly when this folder is absent, for example right after a fresh checkout. **Exclude it from Git.**
- `.sidecars` : local IDE state. **Exclude it from Git.**

Also configure the repository to preserve LF line endings (the format always writes LF); a VCS that rewrites LF to CRLF produces noisy, spurious diffs.

1.5 4. The on-disk format

1.5.1 Project roots

The project *is* a folder; its extension identifies the type.

Extension	Type	CODESYS 3 Binary counterpart
`.fbsproj`	Device project	`.project`
`.fbslib`	Library	`.library`
`.fbsws`	Workspace (for opening the projects)	none
`.fbslibpack`	Library-package archive (single file)	none

1.5.2 Objects and folders

- A programming object is a file named `<ObjectName>.<class>.<language>` : for example a Structured Text function block `Timer` becomes `Timer.fb.st` . The class part encodes the object kind (`prg` , `fb` , `fn` , `meth` , `act` , `trans` , `prop` , `itf` , `struct` , `enum` , `union` , `alias` , `gvl` , ...).
- An object that has children (for example a function block with methods) becomes a directory whose name ends with a caret `^` :

```
Timer.fb.st^/
    Timer.fb.st        (the function block itself)
    Start.meth.st      (a method)
    Stop.meth.st       (a method)
```

- A folder is a directory of the same name plus a `.folder_meta.json` file holding the folder's stable GUID and any properties.
- Project metadata lives in `ProjectInfo.json` (title, version, author, company, description, default namespace, release state, library categories, custom properties).
- A workspace (`.fbsws`) is a small JSON file listing member projects by relative path.

1.5.3 Source-text layout

The source of an IEC object is stored as plain text: no XML header, no binary wrapper. A file contains the object header, the declaration (`VAR … END_VAR` blocks), and the implementation. Object-level data that cannot be reconstructed (such as the stable object GUID) rides along in a leading metadata comment:

```
(* METADATA
{
    "v3Meta": {
        "objectGuid": "e75fc257-..."
    }
}
*)
```

Structured Text is marker-less: declaration and implementation follow directly as plain ST. **All other implementation languages** wrap the body in markers so the surrounding text stays language-agnostic:

```
__BEGIN_IMPLEMENTATION('CFC')
{ ... implementation body in the language's own text format ... }
__END_IMPLEMENTATION
```

1.5.4 File conventions

Every writer in CODESYS 3 FBS honours the same rules. They are what make the files behave in version control:

- **Encoding:** UTF-8 without a BOM, for IEC source, JSON, and XML alike. On read, an existing XML file is honoured per its own prolog, and a BOM is tolerated.
- **Line endings:** always written as Unix LF (`\n`). On read, both LF and CRLF are accepted. *Configure your VCS to preserve LF.*
- **Indentation:** JSON and XML use one tab per nesting level.
- **Determinism:** stable ordering, no timestamps, no absolute paths, so the same project produces the same bytes, and a diff shows only real changes.

1.5.5 Name encoding

File systems and IEC naming rules do not fully overlap, so CODESYS 3 FBS escapes special characters as two underscores plus two hex digits (`.` → `__2e` , `/` → `__2f` , `^` → `__5e`), and wraps Windows reserved device names (`CON` → `_CON_`). Object names are limited to 240 characters, may not contain `#` , may not have leading or trailing spaces, must be case-insensitively unique within a folder, and method overloading is not permitted.

1.6 5. The native CODESYS 3 FBS format

Two categories of content get a clean, human-readable, mergeable representation.

1.6.1 5.1 IEC objects written in Structured Text

For these objects the file body is IEC text: an ST declaration block and, where applicable, an ST implementation body. Out of the box, the following are stored as clean source:

Object	File suffix
Program, Function Block, Function	`.prg.st` , `.fb.st` , `.fn.st`
Method, Action, Transition	`.meth.st` , `.act.st` , `.trans.st`
Interface, Interface Method, Interface Property	`.itf` , `.meth` , `.prop`
Global Variable List (plain)	`.gvl`
DUTs: Struct, Enum, Union, Alias	`.struct` , `.enum` , `.union` , `.alias`
Property (get/set accessor bundle)	`.prop.st`

The declaration part is always Structured Text, regardless of the implementation language. CODESYS 3 FBS owns the object container and the `VAR … END_VAR` blocks; only the *implementation body* is delegated to a language-specific codec. That is why a variable declaration always diffs cleanly even for a graphically implemented POU.

1.6.2 5.2 Well-known project artifacts stored as JSON

A handful of structural artifacts are modelled explicitly as JSON:

Artifact	File(s)
Library Manager	`Libraries.json`
Project Information	`ProjectInfo.json`
Project settings, compiler prefs, lib-dev prefs	dedicated readable mappings
Folder metadata	`.folder_meta.json`

Because these are real JSON with a stable schema, changes to library references, placeholders, or project metadata show up as readable line-level diffs.

1.6.3 5.3 Which IEC *languages* are readable

Language support is codec-based. Structured Text is built in and always available; its body is plain ST source text, marker-less.

The graphical languages, LD, FBD, SFC, CFC, have no built-in codec. Out of the box they are *not* stored as cleanly readable text; they fall back to the XML representation described in the next section. They become readable only if a new language add-on is installed, at which point their bodies are written in that language's own text format between the `__BEGIN_IMPLEMENTATION` / `__END_IMPLEMENTATION` markers.

> *Today, ST-based IEC content (all POU classes, DUTs, GVLs, interfaces, properties) plus Library Manager and Project Info are clean, diffable files. Graphical languages are clean only with an add-on codec installed; the add-ons carrying those codecs are being released step by step.*

1.7 6. The compatibility format

CODESYS 3 FBS is designed not to lose content, but not everything gets a readable form. Addons for CODESYS 3 (e.g. Visualization) need to support CODESYS 3 FBS. If Addons don't support CODESYS 3 FBS, two fallback tiers cover the rest.

1.7.1 6.1 The `.xml.v3` fallback (opaque but lossless)

Anything that has no native CODESYS 3 FBS mapping is stored as a single `<ObjectName>.<type>.xml.v3` file. Inside, two marker sections carry:

- `__V3_META__` : the object metadata and properties, in JSON;
- `__V3_CONTENT__` : the **original, unchanged CODESYS 3 XML serialization** of the object.

Why this tier is opaque — and why that is not a verdict on XML. It would be wrong to read this as "JSON good, XML bad." XML is a fine format for a document with a designed schema, and FBS applies the same output contract to it as to everything else: UTF-8 without BOM, LF, one tab per level, deterministic bytes. The markup language is not the problem, which XML is.

`__V3_CONTENT__` holds the **native CODESYS export format**: the same XML that the export command in the IDE writes for an object and that import reads back, a serialization of the internal object model, made for moving an object between CODESYS installations, not for storing it in a readable form. FBS does not reformat it, it puts that representation in its own file. Its structure therefore follows the implementation rather than the concept, element names and nesting come from the internals of the component that owns the object, and internal identifiers travel alongside the data, so a small edit in the IDE can surface at places in the file that bear no obvious relation to it. There is also no schema to edit against: what is valid is what the reader accepts, which is the real reason hand-editing is unsupported (section 9.4).

So `.xml.v3` is text in the sense that matters for tooling (searchable, Git-safe, lossless) but not in the sense that a human can review it. The native tiers are *designed projections* of the project; this one is a *mechanical dump*. It exists so that no information is lost for objects FBS does not model natively.

What lands here today:

- Graphical POUs (LD / FBD / SFC / CFC) when a language add-on version is installed, that does not support the FBS format.
- Device descriptions, visualizations, task configuration, application objects, cam or cnc objects and essentially every other non-IEC object kind.

> ***Note:** The `.xml.v3` fallback wraps the native CODESYS 3 serialization, so it is understood only by CODESYS 3. CODESYS 4 cannot read `.xml.v3` content.*

1.7.2 6.2 Unknown objects (the load-error safeguard)

This tier applies to an object that CODESYS 3 FBS *recognizes* (a mapping claims the file by its suffix) but cannot **load s**uccessfully. Instead of aborting the load, CODESYS 3 FBS keeps the file as an *unknown object*: the raw content is stored verbatim and surfaced in the IDE with a "could not be deserialized" warning and an "Open in Explorer" action. This happens when loading the object fails, for example:

- parsing or deserialization throws;
- the object's metadata is invalid;
- a duplicate object GUID is detected (a fresh GUID is assigned so the content stays visible rather than being dropped).

The guarantee is that the bytes survive a round-trip, so nothing is silently lost, even though CODESYS 3 FBS could not build a structural model of the object on this load.

1.8 7. Migration: when an object type or product gains CODESYS 3 FBS support

Today an object type may fall back to `.xml.v3`, and later a codec or mapping for it becomes available (for example a new graphical language add-on gets installed, or a product ships native CODESYS 3 FBS support for one of its object kinds). What happens to projects already stored on disk?

CODESYS 3 FBS handles this with in-place migration rather than a re-conversion:

1. When a project is opened, CODESYS 3 FBS notices that an on-disk `.xml.v3` file embeds an implementation or object type that *now* resolves to a registered native codec/mapping.
2. It offers to migrate that file. Migration removes the `.xml.v3` file and re-writes the object in the new native format.
3. Only the on-disk representation changes: the object's identity (its GUID), its place in the tree, and its semantics are untouched. In the diff it appears as "this object is now stored as readable text."

Two consequences for teams:

- **Migration is opt-in and visible.** When a migration is available, a hint appears in the status bar; from there it is triggered through a dialog, not silently on every save, so you decide when the format churn lands in your history.
- **Order the change deliberately.** Because a migration rewrites files, land it in a dedicated commit (ideally on its own branch) rather than mixing it with functional changes: the same discipline you would apply to a formatter or lint-fix sweep.

The general rule: new format support is additive and backward-compatible. Older projects keep working from their `.xml.v3` fallback; when support arrives, migration upgrades them on your terms.

1.9 8. CI/CD with the scripting bridge

Automation stays inside the CODESYS toolchain: CODESYS 3 FBS ships a scripting bridge (`ScriptDriverFBS`) that exposes CODESYS 3 FBS operations to the CODESYS Python scripting engine, which is what makes headless automation practical. A scripting run can:

- **Open** a CODESYS 3 FBS project, library, or workspace folder headlessly (`fbs.open(path, …)`);
- **Convert** an existing CODESYS 3 Binary project to CODESYS 3 FBS (`project.convert_to_fbs(path, …)`);
- **Validate** a project with the CODESYS 3 FBS validators (`project.validate_for_fbs()`): they check for known, common problems, not whether the project is valid. You get back structured results (status, message, affected object) that you can turn into a pipeline pass/fail.

Typical pipeline stages this enables: a quality gate that validates every merge request, an automated conversion step, and build/packaging steps driven from the version-controlled source of truth.

> *Note: the scripting bridge drives the **CODESYS scripting engine**; it is headless in the sense of "no interactive UI," not "no CODESYS installed." A build agent still needs a CODESYS 3 installation with the CODESYS 3 FBS add-on and a valid license. See section 9 for what this implies.*

1.10 9. External tooling and the "open format" question

The format is open and documented. At the same time, CODESYS 3 FBS was not developed to support external tooling. Both are true; this section makes the boundary precise. External tools can read the storage freely, and they can write the text-modeled subset within the rules below; but editing belongs in CODESYS, and version control belongs to CODESYS Git 2.0 (section 3).

1.10.1 9.1 What "open" actually means here

Openness in CODESYS 3 FBS means transparency and tooling-friendliness for the software lifecycle: diffing, merging, reviewing, branching, searching, validating, and automating. The project is legible to Git, to CI, and to a human reviewer.

Openness does not mean "the on-disk files are a free-form editing surface where any change you type is guaranteed to be a valid CODESYS project." A CODESYS project is a structured object model with cross-references (by GUID), consistency rules, and semantics that the IDE and compiler enforce. CODESYS 3 FBS is a faithful, transparent *projection* of that model, not a replacement for it.

This leads to two rules:

- **Reading** the storage from outside CODESYS is supported: that is what Git, reviewers, and CI do.
- **Writing** the storage from outside CODESYS is possible for the text-modeled subset, but the result is a valid project only once CODESYS has loaded and built it; the CODESYS 3 FBS validators catch known problems but do not prove validity.

1.10.2 9.2 What external tools can do

Within that boundary, read-mostly external tooling works well, because the readable subset is plain text and JSON with documented conventions:

- **Search and navigation** across the whole project with ordinary tools (`grep` , ripgrep, editor project-wide search).
- **Static checks and metrics** on ST source, or on `Libraries.json` / `ProjectInfo.json` (for example, "does any project use a forbidden library version?").
- **Code generation** that emits ST or DUT files following the naming and format conventions.
- **Documentation extraction** from declarations and comments.

One rule makes all of this safe: respect the output contract (UTF-8 no BOM, LF, tabs, the metadata comment, container `^` directories) and do not touch the opaque tiers (`.xml.v3` , unknown objects). Anything you write should be checked by opening it in CODESYS, the authority on whether it is a valid project; the validators in section 8 additionally flag known problems.

1.10.3 9.3 Long-term readability

Text-and-JSON storage is also an archival benefit: the readable subset of a project remains inspectable with nothing more than a text editor, independent of whether a matching CODESYS 3 Binary reader is at hand. The `.xml.v3` fallback keeps the rest lossless.

1.10.4 9.4 Concrete "do not" list

- **Do not hand-edit the opaque tiers.** `.xml.v3` files are native CODESYS 3 XML. Unknown objects are verbatim blobs. Editing them by hand is unsupported and likely to corrupt the object.

- **Do not rewrite GUIDs or the `(* METADATA *)` comment by hand.** Object identity is what keeps renames, moves, and cross-references stable; breaking it turns a clean diff into a delete-plus-recreate and can break references.
- **Do not restructure the tree in a file manager (renaming container `^` directories, moving objects between folders, renaming files) and expect the semantics to follow.** Do these operations in CODESYS so identity and references are maintained.
- **Do not edit the folder while the project is open in CODESYS.** There is no live synchronization between an open instance and the disk. See section 12.
- **Do not commit derived data** (see section 3.4).
- **Do not assume a hand-written change is valid until it round-trips.** Open and build the project in CODESYS before trusting an externally produced edit; the CODESYS 3 FBS validators flag known problems but do not confirm validity.
- **Do not ignore line-ending configuration.** A VCS that rewrites LF to CRLF will produce noisy, spurious diffs.

1.11 10. Using AI harnesses on the storage without the CODESYS 3 IDE

Can an AI coding agent (an LLM-based harness) read and modify a CODESYS 3 FBS project directly on disk, without the CODESYS 3 Development System in the loop? Section 9 already answers this: it works within clear limits, while editing and version control remain with CODESYS and CODESYS Git 2.0.

FBS ships nothing for this. No skills, no prompt packages, no agent instructions. A harness learns the format from this documentation and from the files themselves.

Reading: yes, and it is a strong fit. The readable subset (ST POUs, DUTs, GVLs, interfaces, `Libraries.json`, `ProjectInfo.json`) is the kind of content an AI harness works well with. An agent can search the tree, explain a function block, spot smells in ST code, summarize a library manifest, or generate review comments, all from the files alone.

Writing: possible for the text-modeled subset, behind a CODESYS check. Because ST source round-trips as plain text, an agent can propose edits to declarations and ST implementations, generate new ST objects following the naming and format conventions, or refactor across many files. To be safe it must:

- **Stay inside the readable subset.** Never modify `.xml.v3` files or unknown objects; never invent or rewrite GUIDs; preserve the `(* METADATA *)` comment.
- **Honour the output contract.** UTF-8 without BOM, LF line endings, tab indentation for JSON, correct file naming and container `^` directories.
- **Not restructure the tree blind.** Renames/moves that change identity or references should go through CODESYS, not a raw file rewrite.
- **Close the loop with CODESYS.** Every change must be verified before it is trusted; the practical mechanism is the same headless path CI uses: open and build the project via the scripting bridge, and run the CODESYS 3 FBS validators for an extra check of known problems (see section 8). Loading is what decides validity; without it, an agent can produce files that *look* right but are not a valid project.
- **Only write while the project is not open in CODESYS.** An open instance neither picks up external file changes nor keeps the disk current between saves (see section 12); changes written in parallel with an open IDE session can be overwritten by the next save.

1.12 11. Licensing and read-only mode

CODESYS 3 requires a valid CODESYS 3 FBS license, the **Professional Developer Edition**, to *save* file-based projects. Without it, a CODESYS 3 FBS project still opens, but read-only:

- Reading, diffing, and reviewing work as usual; it is plain file access, so Git, external tools, and reviewers are unaffected.
- The Save command is disabled, and saving through CODESYS is cancelled.
- The project becomes writeable again as soon as a valid license is present.

A missing license therefore limits *writing through CODESYS*, not your ability to read the project or work with it in version control.

1.13 12. Limitations

CODESYS 3 FBS does not currently synchronize between an open CODESYS instance and the project folder on disk:

- **External changes are not detected.** CODESYS does not monitor the project folder with a file watcher. If project files are modified on disk while the project is open in CODESYS, the changes are not detected and the affected objects are not reloaded. The next save writes the in-memory state back to disk and can overwrite the external modifications.
- **Editor changes are not written through to disk.** Changes made in the CODESYS editors are held in the in-memory project model and are written to the CODESYS 3 FBS files only when the project is saved. Between saves, the on-disk state does not correspond to the state in the IDE, and file watchers of external tools are not notified.

Concurrent editing by an open CODESYS instance and another tool on the same project folder is therefore not supported. The on-disk state is synchronized only at load and save. Recommended procedure:

- Close the project in CODESYS, or save it and make no further edits, before an external tool writes to the project folder.
- Reopen the project in CODESYS after external changes to load them.
- Read-only access (search, review, analysis; see section 9.2) is not affected; it reflects the state of the last save.

1.14 Appendix A: Format reference

1.14.1 File and folder types

Name / pattern	Meaning
`*.fbsproj`	Device project (folder)
`*.fbslib`	Library (folder)
`*.fbsws`	Workspace file (JSON)
`*.fbslibpack`	Library-package archive (single file)
`<Name>.<class>.<language>`	Programming object file (e.g. `Timer.fb.st`)
`<Name>.<class>.<language>^/`	Container object directory (has children)
`<Name>.<type>.xml.v3`	Opaque V3-XML fallback for non-native objects
`.folder_meta.json`	Per-folder metadata (GUID, properties)
`ProjectInfo.json`	Project metadata
`Libraries.json`	Library Manager

Name / pattern	Meaning
`.auxiliaries`	Regeneratable cache, exclude from VCS
`.sidecars`	Local IDE state, exclude from VCS

1.14.2 Output contract

Aspect	Rule
Encoding	UTF-8 without BOM (BOM tolerated on read)
Line endings	Write LF; read LF or CRLF
Indentation	One tab per nesting level (JSON, XML)
Determinism	Stable order, no timestamps, no absolute paths

1.14.3 Implementation-body markers

```
__BEGIN_IMPLEMENTATION('<language>')
... body in the language's own text format ...
__END_IMPLEMENTATION
```

Structured Text is the exception: it is written marker-less as plain source.

1.15 Appendix B: Glossary

- **CODESYS 3**: the current CODESYS generation (3.5), as a product: IDE, compiler, project model.
- **CODESYS 3 Binary**: the classic monolithic binary storage format of CODESYS 3 (`.project`, `.library`, `.projectarchive`). Opaque to version control.
- **CODESYS 3 FBS**: File-Based Storage; the folder-based storage format for CODESYS 3 projects described in this document.
- **CODESYS 4 projects**: projects of the next-generation CODESYS product, which stores its projects file-based natively and originates the file-based text format. Has no object GUIDs or properties, and may use features CODESYS 3 cannot represent (relative library references, semantic-version placeholders); CODESYS 3 opens such projects read-only.
- **file-based text format**: the on-disk text format shared by CODESYS 3 FBS and CODESYS 4.
- **CODESYS Git 2.0**: the CODESYS Git integration, version 2.0.0.0 or newer; builds on CODESYS 3 FBS and is the recommended version-control mechanism for CODESYS 3 FBS projects.
- **Codec**: an add-on component that owns the *implementation body* of one IEC language, replacing the `.xml.v3` fallback for that language with readable text.
- **Object mapping**: an add-on component that owns a whole *object type* and its file, replacing the `.xml.v3` fallback for that type.
- **Auxiliary**: a regeneratable cache entry, never project content; missing auxiliaries are rebuilt, never treated as an error.
- **Unknown object**: a file CODESYS 3 FBS cannot interpret, preserved verbatim so nothing is lost.
- **Container object**: an object with children, stored as a `^`-suffixed directory.