CODESYS

Whitepaper

AI Meets IEC 61131-3

Standardized Integration of Large Language Models into the CODESYS Development System

Programmable Logic Controller (PLC) projects are growing in complexity while engineering timelines are shrinking. At the same time, Large Language Models (LLMs) have demonstrated substantial capability in software code generation, yet they lack awareness of proprietary project structures, libraries, and conventions specific to industrial automation. The CODESYS Development System MCP Server addresses this gap by implementing the Model Context Protocol (MCP), an open standard for communication between AI models and external applications. Through a defined set of tools, the MCP Server gives LLMs direct, structured access to CODESYS projects, enabling them to read, generate, and modify Structured Text code, query library documentation, and use compiler and static analysis feedback, without requiring copy-and-paste workflows or proprietary integrations. This paper explains the architecture, practical applications, design rationale, and security considerations of this approach, and outlines its implications for the daily work of control application engineers.

Introduction

The automation industry is witnessing a paradox. On the one hand, PLC applications now routinely encompass thousands of variables, dozens of function blocks, and deeply nested state machines that would have been unthinkable a decade ago. On the other hand, the time allocated for engineering these applications is compressed by shorter machine development cycles and increasing competitive pressure. Established features of the CODESYS Development System (automatic syntax completion, the Professional Developer Edition's CODESYS Static Analysis for enforcing coding guidelines, the CODESYS Test Manager for automated testing, and the Application Composer's no-code module approach) already support efficient project engineering. Yet the fundamental constraint remains: it is the programmer who, based on experience and available code libraries, translates the automation task into working logic.

Generative AI, specifically Large Language Models, has entered this equation with considerable momentum. LLM vendors claim that within a few years, traditional manual coding will be marginalized. And indeed, the results in many domains of software development are impressive. But how applicable are these models to the very specific domain of PLC programming against the IEC 61131-3 standard? Where do they genuinely help, and where do they introduce risk? An LLM such as Claude or ChatGPT is trained on publicly available data. It therefore possesses extensive knowledge of general programming patterns but has no awareness of the internal structure of a running CODESYS project, the specific library repository of a particular OEM, or the architectural conventions a team has established. Without this context, the model fills the gaps with statistical probabilities, producing not syntax errors, but the kind of subtle, structural misjudgements that are discussed in detail later in this paper.

This is precisely the problem that the Model Context Protocol addresses. MCP, originally developed by Anthropic and now maintained as an open standard with broad ecosystem support from clients including Claude, ChatGPT, GitHub Copilot, and Cursor, defines a structured communication pathway between an AI model and an external application. Think of it as a standardized connector, analogous to USB-C for peripheral devices, that allows any MCP-compatible AI assistant to access tools, query data, and perform actions within an application, without requiring a customized integration for every model-application combination.

Prior to MCP, integrating AI capabilities into existing engineering tools was expensive and proprietary. Every vendor developed their own integration, there were no standardized interfaces. MCP eliminates this fragmentation. It follows a client-server architecture: an AI model host (MCP client) connects to one or more MCP servers, each of which exposes a defined set of tools the model can use. The protocol handles capability negotiation, transport, and data serialization under the hood, but the practical consequence is what matters: users can switch the underlying LLM without rewriting the integration logic, and the same MCP Server can serve different AI services without modification.

The CODESYS Development System MCP Server implements this protocol as a product add-on integrated into the IDE. It provides the command infrastructure of the CODESYS Development System as MCP tools, giving any connected LLM structured access to project contents, programming objects, libraries, compiler diagnostics and more. The critical difference from a simple chat interface or clipboard-based workflow is directness: the model reads from and writes to the live project. There is no need to export code, paste it into a chat window, receive modified code in return, and paste it back. The model operates within the project context, and every change it makes is immediately visible in the open editor, where the engineer can check it.

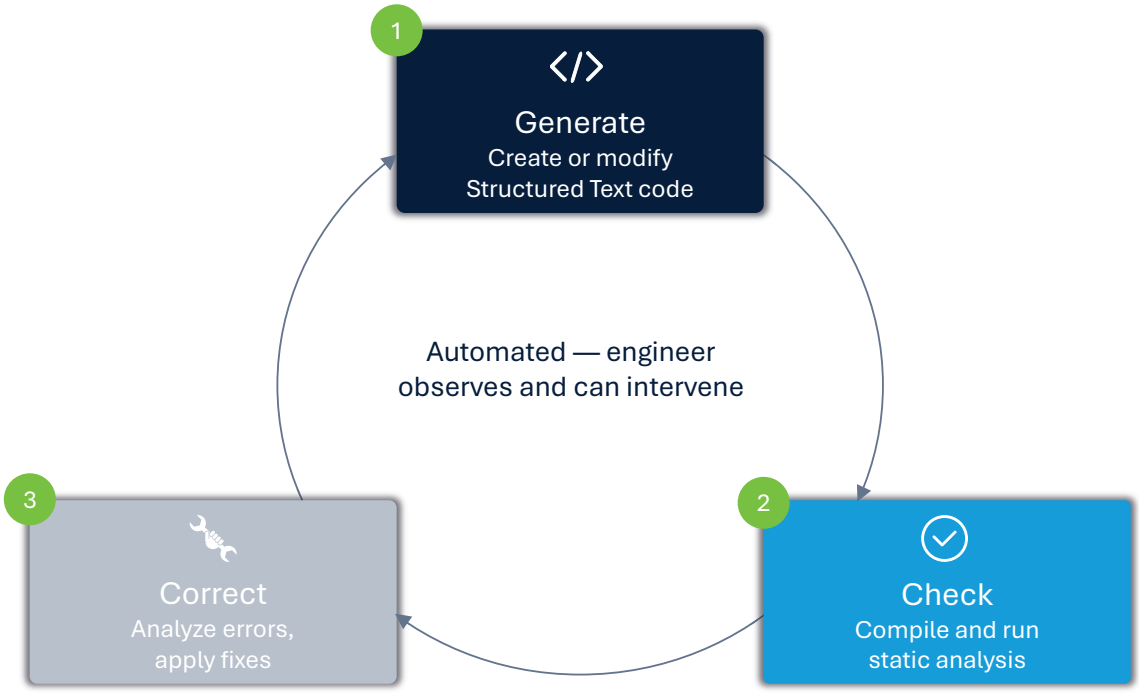
How the CODESYS Development System MCP Server works in practice

The MCP Server provides a set of tools, each mapping to a specific IDE operation. These tools divide into two categories, read-only and write, and understanding this distinction is essential for engineers who want to maintain control over what the AI does in their project.

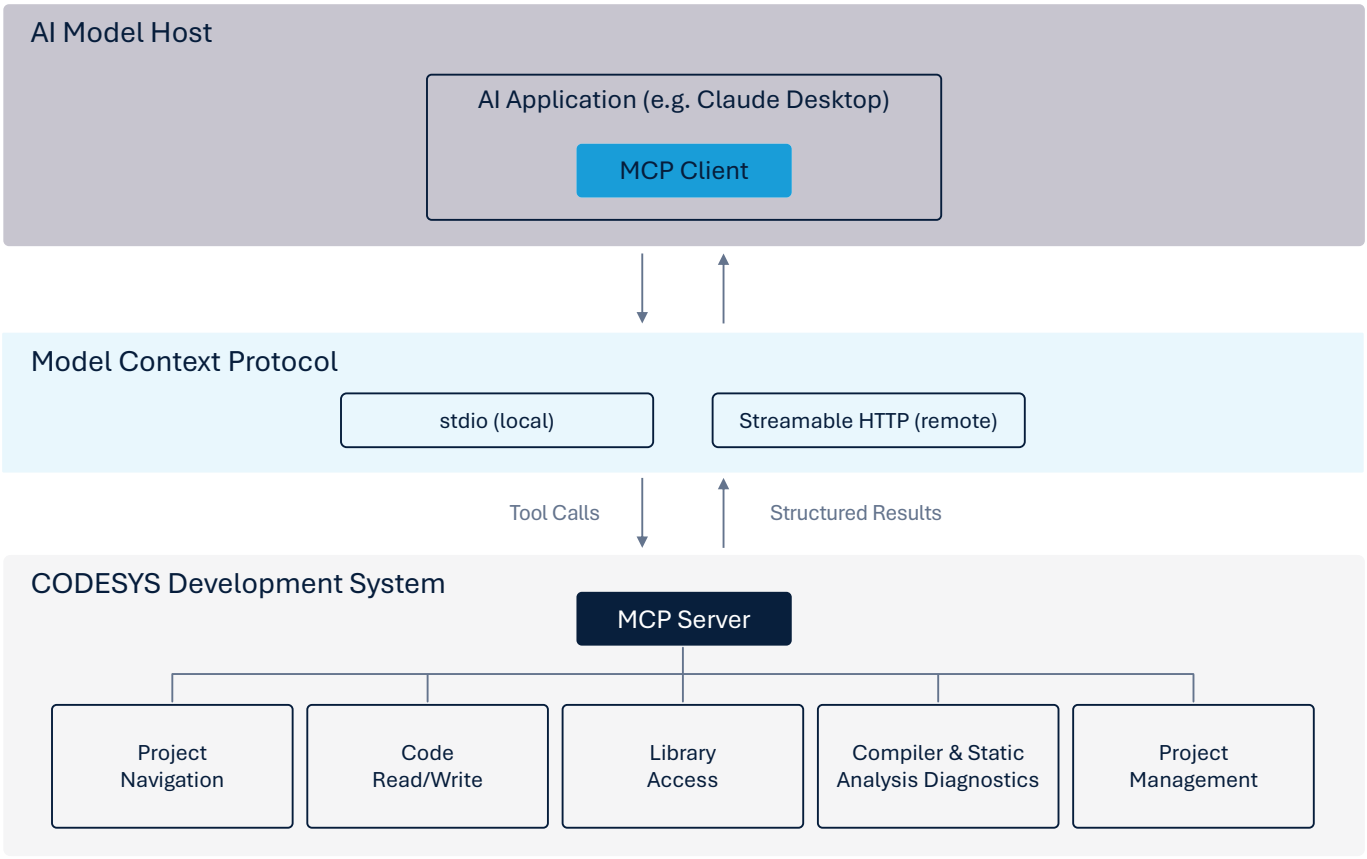
On the read side, the model can browse the project tree, read Structured Text source code, search across the project by pattern or regular expression, and retrieve the device tree and I/O configuration. A dedicated group of library tools gives the model access to the complete library landscape, from high-level overviews down to parameter-level documentation for individual functions, always reflecting the user's actual library environment without prior training or manual configuration. This eliminates one of the most common sources of hallucination in LLM-generated PLC code: guessing at function signatures or inventing library calls that do not exist. The compiler feedback tool closes the loop by returning diagnostic messages with source paths and line numbers, enabling the model to check its own output and iterate on errors automatically.

On the write side, the model can create or rewrite Structured Text objects (including complete POUs with nested sub-POUs in a single operation), perform targeted text replacements, manage folders and library references, and assign programs to task configurations. A full reference of all available tools, their parameters, and their effects is provided in the CODESYS Development System MCP Server online documentation.

What makes this toolset significant is the feedback loop it enables. Consider a concrete scenario: an engineer instructs the AI to „create a function block for two-point control with hysteresis and instantiate it in the main program." The model uses *browse_project_tree and get_libraries_referenced_in_application* to understand the current project structure and available resources. It then calls *create_or_replace_structured_text_object* to produce the function block with its declaration and implementation. It uses *add_program_call_to_task* or modifies the main program via *replace_text_in_structured_text* to insert the instance call. It then runs *check_for_errors* to verify compilation. If compiler errors or CODESYS Static Analysis violations are returned, the model analyzes the diagnostic messages, which include source paths and line numbers, and makes corrections, re-checks, and iterates until the code compiles cleanly and passes the configured analysis rules. This generate-check-correct cycle runs automatically, and the engineer observes every step in the IDE, able to intervene at any point. This is fundamentally different from asking an AI chat to write code in isolation: the model works against real project data, real libraries, and a real compiler, not against assumptions about what the project might contain.



The same tools serve a very different workflow when applied to existing code rather than new code. An engineer inheriting an unfamiliar project can ask the AI to explain its structure and identify potential issues. The model uses *browse_project_tree* to map the overall architecture, reads individual POUs with *get_structured_text_content*, queries the library documentation to understand which external functions are in use, and cross-references the device and I/O configuration. On this basis, it can produce a structured summary of the project's logic, flag areas of high complexity or unusual patterns, and suggest where documentation is missing, all without modifying a single line of code. For experienced engineers reviewing a colleague's work or onboarding onto a legacy system, this read-only analysis workflow delivers immediate value with zero risk to the project. To do this, just disable the MCP Tools with write access in the AI tool.



What determines the quality of AI-generated code

The quality of AI-generated results depends far less on the sophistication of the prompt than on the quality of the environment the model works in. Three factors matter most: the availability of precise and task-relevant context, the enforcement of coding rules through static analysis (for which CODESYS provides the CODESYS Static Analysis tool), and the presence of tests that verify functional correctness, supported by the CODESYS Test Manager.

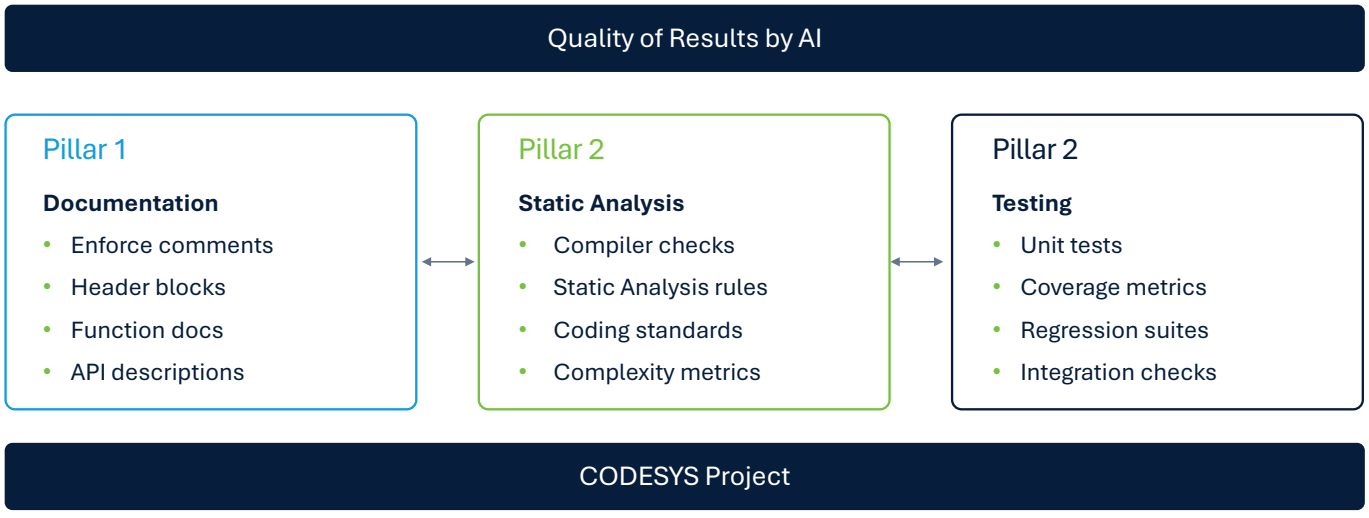
Context, in this regard, means not the maximum possible amount of information but a small, precise, and task-related subset. Overloading a model with irrelevant context makes it imitate irrelevant patterns rather than solving the specific task at hand, thus degrading the output. The CODESYS Development System MCP Server addresses this problem by delivering focused, pre-processed information (a specific POU's source code, a specific library function's signature, a specific compiler error including location) rather than dumping the entire project into the model's context window. This design aligns with established principles for effective tool design in agentic AI systems, as described in Anthropic's „Writing effective tools for agents – with agents," where tool responses should prioritize contextual relevance over flexibility and return only high-signal information. It also makes sense to maintain a modular documentation in a markdown structure and as inline comments in the code.

Documentation, static analysis, and testing function as reinforcing pillars. Documentation is the structural framework: it defines what applies, and the model follows it. But documentation alone cannot guarantee that every output conforms to the stated rules.

The CODESYS Static Analysis tool, part of the Professional Developer Edition, offers a comprehensive rule and metrics framework (more than 100 configurable rules, PLCopen Coding Guidelines sets, naming conventions, and over 20 quality metrics), but what matters for AI-assisted workflows is the integration: when configured to run automatically after each compilation, its diagnostic messages reach the AI through the same *check_for_errors* tool used for compiler diagnostics. The AI does not merely compile the code; it checks the code against the team's actual quality standards and corrects violations in the same iterative loop.

This enforcement becomes even more critical in AI-assisted workflows. Architecture rules, dependency directions, layer separation, and coding style conventions that exist only as verbal agreements or Wiki entries erode faster under AI-generated code than under human-authored code, precisely because the model takes the path of least resistance. If a problematic pattern is possible, it will eventually appear.

Testing provides the final verification. Documentation tells the model what to do. Static analysis prevents structural violations and enforces coding guidelines. Tests verify whether the result is functionally correct. In AI-assisted development, high test coverage is not optional but a prerequisite; the complexity of AI-generated changes regularly exceeds what a human review alone can reliably validate. Tests also serve a dual function: they secure correctness, and they serve as specification. A well-written test describes expected behavior more precisely than any prose requirement.



The same principle applies to coding conventions and documentation at project level. Clear variable names, consistent naming guidelines, comprehensible comments, and well-documented library functions measurably improve the model's output quality. Conversely, if a codebase is poorly documented and inconsistently structured, the model will reflect those weaknesses in its generated code. Internal piloting has revealed specific, recurring failure patterns in AI-generated Structured Text code that engineers should learn to recognize:

- Redundant variable initializations scattered across multiple POUs rather than centralized in a Global Variable List or parameter interface. This creates maintenance hazards because a single parameter change must be propagated to every location manually, and any missed instance becomes a latent bug that may not surface until runtime.

- Incorrect reuse of library function blocks with wrong parameterization because the model guessed at signatures instead of querying the library documentation. The resulting code compiles if the parameter types happen to match, but the behavior diverges silently from what the library author intended.

- Duplicated control logic across several programs where a shared function block should have been extracted. Over time, the duplicates drift apart as individual instances are modified, making the codebase increasingly difficult to maintain and test.

- Configuration constants hard-coded directly in implementation code rather than exposed through a parameter interface. This buries variable parameters deep in the logic, where they are invisible to commissioning engineers and inaccessible to higher-level parameterization tools.

Each of these patterns is a concrete candidate for a CODESYS Static Analysis rule or test, and awareness of them helps engineers conduct more targeted reviews of AI-generated output. This leads to a useful heuristic: if the AI cannot understand a project, new team members and external service providers will struggle with it, too. The model, in this sense, acts as a measure for the inherent comprehensibility of a project.

The engineer's role: steering, not surrendering

The engineer's role shifts but does not diminish. It moves from writing every line manually to a steering function: specifying tasks precisely, evaluating results critically, and feeding corrections back into the working environment. Notably, the investment this requires (documentation, static analysis rules, and tests) pays off regardless of whether AI is used. The introduction of AI-assisted workflows merely increases the return on that investment because the model benefits from the same structures that help human developers. In practice, engineers should start with tasks that deliver quick results with low preparation (generating unit tests for known interfaces, documenting legacy code, establishing standard control patterns) and progress toward tasks that require more upfront investment in decomposition and documentation. When results are unsatisfactory after one or two attempts, switching to a more capable model or restructuring the available context is generally more productive than refining the prompt.

Special caution is required for destructive operations. AI tools, particularly when operating in a rather autonomous mode, can execute actions that are not easily reversible. Engineers should be attentive to two risk patterns: first, write operations on project objects are executed immediately (*remove_object*, for example), which is why version control with file-based project storage is strongly recommended before working with AI assistance; second, AIs tend to modify test assertions or expected values to make tests pass, rather than fixing the underlying bug, which makes vigilance around test corrections essential.

The CODESYS Development System MCP Server is designed to give users the full choice of LLMs. The Server has been validated with Anthropic Claude and OpenAI GPT. Other MCP-compatible LLMs work as well. CODESYS also provides context files containing CODESYS-specific domain knowledge (including Structured Text language conventions, common architectural patterns, and guidance for working with MCP tools) that users can feed into their chosen model as reference material. This complements the dynamic context that MCP tools provide automatically: while the library tools always reflect the user's actual, current library environment without any configuration, the context files supply the broader engineering knowledge that helps the model produce idiomatic, well-structured code. The responsibility for selecting and configuring the model, and for making context files available, rests with the user. This preserves full flexibility while ensuring that the model benefits from CODESYS-specific expertise. As MCP is supported by a wide range of clients and servers, including development environments like Visual Studio Code and Cursor, engineers can use the same model with multiple MCP servers simultaneously. A model connected to the CODESYS Development System MCP Server and, for example, a file system server or a version control server can draw on context from multiple sources in a single workflow.

INFO: Security and Data Sovereignty

When project data is sent to a cloud-based LLM for code generation, information security is of the essence. For organizations whose machine designs and source code are too sensitive to leave the corporate network, CODESYS is collaborating with Intel on a local alternative. Using Intel's OpenVINO runtime, an open-source AI inference framework, language models are fine-tuned for PLC code generation and optimized for local execution. They operate without any cloud connection. OpenVINO runs on a wide range of CPU architectures, from Core processors to Xeon server platforms, scaling performance according to the available hardware. Processors with discrete GPUs, integrated GPUs (iGPUs), and Neural Processing Units (NPUs), such as Intel's Core Ultra family, deliver significantly higher AI inference performance, but are not a prerequisite; the same models run on any supported processor. This collaboration focuses specifically on AI inferencing rather than training: the models are optimized, converted, and serialized through the OpenVINO pipeline, then executed entirely on-premises. Prompts and source code remain within the organization, while programming assistance remains available. As the local models also connect to the CODESYS Development System via MCP interface, organizations can use cloud-based and local models interchangeably according to their security requirements. With the release of the CODESYS Development System MCP Server local models can be used immediately.

Regardless of whether an external or local model is used, version control is strongly recommended. CODESYS 3 provides only limited options for undoing changes. File-based project storage together with a version control system such as Git allows engineers to track which changes were made manually and which were generated by AI, providing a complete audit trail. The traceability of Git commit logs is essential both for quality assurance and for regulatory compliance in safety-relevant applications.

Current scope and roadmap

The current release of the CODESYS Development System MCP Server supports Structured Text exclusively for code creation and modification. Objects written in graphical languages such as Ladder Diagram, Function Block Diagram, or Sequential Function Chart can be read, allowing the AI to understand their logic, but not created or modified by the AI. This is a deliberate scoping decision for the initial release. Similarly, the MCP Server currently does not generate visualizations, although HTML5 visualization elements can be handled with some limitations. The CODESYS Development System cannot yet be used in a fully agentic mode where the AI executes multi-step workflows autonomously without human confirmation at each step, as this would require file-based project storage. CODESYS 4 is designed to support this capability natively. For tasks that require graphical language programming, device configuration, or visualization design, engineers work directly in the CODESYS IDE as before.

Looking ahead, the development roadmap extends the scope considerably beyond Structured Text. Planned tools include support for Ladder Diagram, which will open AI assistance to the large share of IEC 61131-3 users who work primarily in graphical languages rather than Structured Text, as well as device configuration generation and parameterization, visualization generation, and online tools for monitoring and debugging. The last of these is particularly significant: online capabilities would allow the AI to assist not only during development but also during commissioning and diagnostics, where rapid interpretation of live data is often the bottleneck. Fully agentic workflows, enabled by file-based storage in CODESYS V3 and natively in CODESYS 4, will allow the AI to execute multi-step tasks autonomously while the engineer reviews the result rather than confirming each individual step.

The CODESYS Development System MCP Server was released as a product add-on for CODESYS V3.5 SP22 on April 28, 2026. It is licensed as part of the Professional Developer Edition, meaning existing subscribers gain immediate access. As the integration is based on the open MCP standard rather than a proprietary protocol, the commitment is structurally protected against changes in the LLM market. Should a better model appear tomorrow, it will connect through the same interface.

Conclusion and outlook

The CODESYS Development System MCP Server is the beginning, not the endpoint. As the scope expands along the roadmap outlined above (from Structured Text to graphical languages, device configuration, visualization, online diagnostics, and eventually fully agentic workflows), the range of engineering tasks that benefit from AI assistance will grow accordingly. Meanwhile, the broader ecosystem is growing: partners such as KS Solutions provide AI-based code conversion between PLC standards, the CODESYS Online Help chatbot answers technical questions exclusively from official documentation, and a local code-completion prototype already generates context-aware multi-line suggestions from up to 200 lines of surrounding program logic.

It's a consistent pattern that ties these developments together. The same generate-check-correct cycle demonstrated earlier for code generation applies equally to test generation, and the interesting shift is in who is verifying whom. Here the AI is not generating application logic; it is writing tests that verify someone else's code. The MCP Server provides the context; the compiler and static analysis provide the guardrails; tests provide the verification. These three pillars work in concert, and the engineer reviews the result rather than assembling it from scratch.

The underlying thesis is simple. Clean documentation, enforced static analysis rules, comprehensive tests, and consistent version control have always been good engineering practice. AI does not change what good engineering is. It changes how much of the routine work stands between engineers and really interesting problems.

CODESYS®

One software. All applications.
Your automation solution.

References and Notes

Model Context Protocol specification and documentation
https://modelcontextprotocol.io

CODESYS Development System MCP Server tool reference
https://content.helpme-codesys.com/en/CODESYS%20Development%20System%20MCP%20Server/_idemcp_mcp_tools_and_ressources.html

CODESYS Online Help chatbot
https://www.helpme-codesys.com/

CODESYS Static Analysis product information
https://store.codesys.com/de/codesys-static-analysis.html

CODESYS Test Manager product information
https://store.codesys.com/de/codesys-test-manager.html

Anthropic, „Writing effective tools for agents — with agents," published September 11, 2025
https://www.anthropic.com/engineering/writing-tools-for-agents