Qt Group

WHITEPAPER

AI-Powered Software Development, Done Right

Best Practices for Code Quality in AI-Augmented Workflows

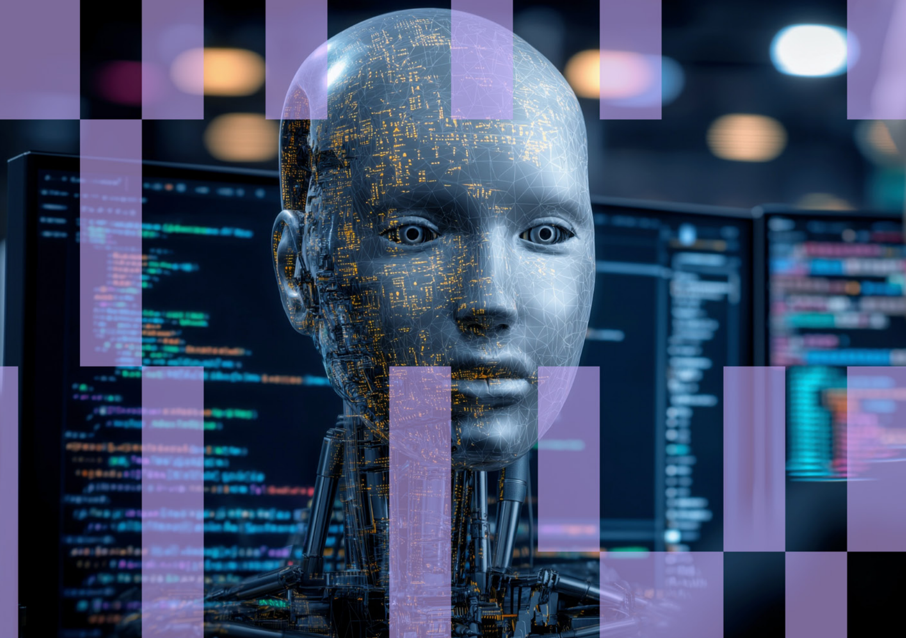


Table of Contents

Introduction 3

AI in Software Development: The Shift. The Promise. The Reality. 4
- The Shift: AI as part of the daily toolchain 4
- The Reality: more code is not the same as more correct code 4
- The Complement: deterministic verification alongside probabilistic generation 4

Axivion in Your AI-Powered Workflow 4
- Same rules, no exceptions 4
- Axivion as the semantic backbone for AI tools 4
- Axivion as a retrieval source: RAG for code 5

Two Levels of Integration: Interactive and Agentic Autonomous Use 6
- Level 1: MCP integration — Axivion as a tool for any AI assistant 6
- Level 2: The VS Code plugin: Axivion native in the IDE 7

Best Practices: Working With AI and Axivion 8
- Run Axivion on every AI-generated change 8
- Feed Axivion findings into your prompts 8
- Prompt with violation context, not blind 8
- Constrain the assistant with explicit rules 8
- Do not relax quality rules for AI code 8
- Know what each side catches 8

The Developer Is Still in Charge 9

Conclusion 9

Introduction

Software development is accelerating. AI assistants now generate code, scaffold tests, and suggest refactors at a speed no individual developer can match. The productivity gains are real , but so are the risks.

AI generates code probabilistically. It is trained to produce output that looks right, not necessarily output that actually is right. In everyday commercial software, the difference may be manageable. In safety-critical systems, such as automotive, aerospace, medical, industrial, it is not. A codebase that violates coding guidelines, drifts from its intended architecture, or contains unreachable logic does not become safer because an AI wrote it faster.

This whitepaper is for engineering teams navigating that gap: organisations that want the productivity benefits of AI-assisted development without compromising the code quality, standards compliance, or audit readiness their industries require.

The answer is not to reject AI tooling. It is to pair it with deterministic verification. Static code analysis and architecture verification do not compete with AI assistants. They complete them. AI proposes; verification decides. This paper sets out what that combination looks like in practice: how Axivion integrates with AI-assisted workflows, at what levels, and what discipline is required to keep the evidence base intact when code volume increases and its origin becomes harder to trace.

This shift does not diminish the developer's role — it redefines it. As AI handles more of the mechanical work, the developer's attention moves up the stack: from authoring syntax to making architectural decisions, from writing boilerplate to judging whether the behaviour the code implements is the behaviour the system actually needs. The volume of code a developer oversees increases; so does the importance of each decision they make about it.

In safety-regulated industries, that human role is not optional. Functional-safety standards assign responsibility and liability to people and organisations, not to tools. No AI assistant can sign off on a release, defend a design decision in an audit, or carry the legal weight of a field failure. What deterministic verification does is keep the developer informed: surfacing findings early, grounding AI suggestions in the real state of the codebase, and ensuring that the evidence base required for certification remains intact regardless of how the code was produced. The developer stays in charge. The tools make that a tractable position to hold.

The core principle is simple: the origin of a line of code does not change the obligation to verify it.

AI in Software Development:
The Shift. The Promise. The Reality.

The Shift: AI as part of the daily toolchain

AI assistants have become a routine part of how developers write software. Code completion, refactoring suggestions, test scaffolding, and full-function generation are no longer experimental; they sit inside the IDE alongside the compiler. The measurable effect is a sharp increase in throughput: more code, written faster, across more of the day-to-day tasks that used to consume engineering time.

The Reality: more code is not the same as more correct code

That throughput comes with a caveat. AI-generated code is not inherently correct, safe, or compliant. Large language models produce plausible output, not verified output, and they hallucinate APIs, misuse memory, and reproduce patterns that violate coding standards. The volume and velocity of generation amplify the problem: more code in less time means more potential defects entering the codebase before any human has read them carefully. The developer's role is shifting accordingly: from author to reviewer, integrator, and arbiter, but it is not disappearing. Engineering judgment, architectural intent, and accountability for the final artifact remain human responsibilities.

The Complement: deterministic verification alongside probabilistic generation

Static code analysis and AI assistance are complementary, not competing. AI is pattern-based and probabilistic; static analysis is deterministic, repeatable, and auditable. One proposes, the other verifies. This complementarity leads to the principle that anchors the rest of this whitepaper: AI-generated code must be treated exactly like human-written code. The same rules, the same gates, the same evidence. Origin does not change obligation.

Axivion in Your AI-Powered Workflow

Same rules, no exceptions

The starting point is operational, not philosophical: every piece of AI-generated code that lands in the repository runs through the same Axivion analysis as code a human wrote. The coding standards (MISRA, AUTOSAR C++14, CERT, custom rule sets) and the architecture rules apply uniformly. Quality gates block commits or merges that violate them, regardless of who or what produced the lines. This matters most in safety-regulated industries. Functional-safety standards such as IEC 61508, ISO 26262, IEC 62304, and EN 50716 require traceable evidence that code conforms to its rules, independent of how it was authored. The forthcoming ISO/PAS 22440, which addresses AI in safety-related software, is still under development; until it stabilizes, the pragmatic path is to preserve audit readiness today by treating AI output as in-scope for the same verification regime as the rest of the codebase.

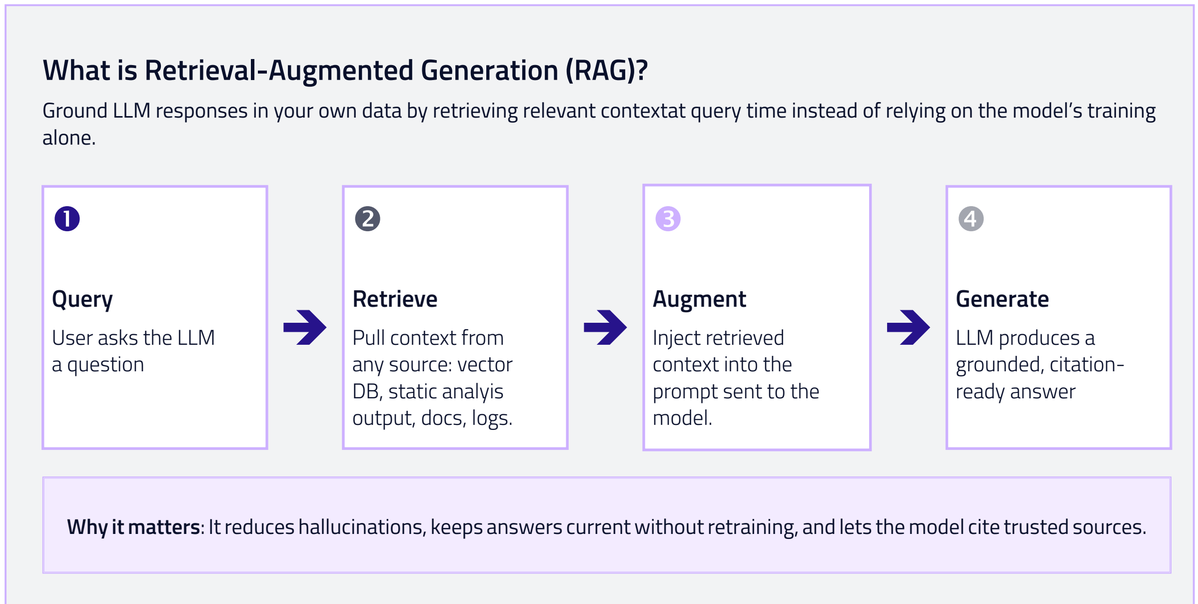
Axivion as the semantic backbone for AI tools

Static analysis is not only a downstream check. Axivion maintains a precise, global model of the codebase: symbols, call graphs, type information, architecture layers, applied rules, and active findings. That model is exactly the kind of grounded, project-specific context that LLM-based assistants need but rarely have. Exposed to an AI assistant, it replaces guesswork and grep-style heuristics with semantic facts about the actual project: where a function is called from, which architecture layer a module belongs to, which rule a given line violates and why. The result is better suggestions and fewer fabrications, because the model now reasons over verified information rather than inferences drawn from surrounding tokens. In this role Axivion also acts as a guardian for the AI itself; a deterministic check that catches what the probabilistic side gets wrong before it reaches a human reviewer.

Axivion as a retrieval source: RAG for code

The mechanism that connects Axivion to an AI assistant is retrieval-augmented generation (RAG). Rather than retraining a model on a specific project, which is costly, slow, and quickly stale, RAG retrieves relevant context at query time and injects it into the prompt. The model then generates an answer grounded in that retrieved evidence.

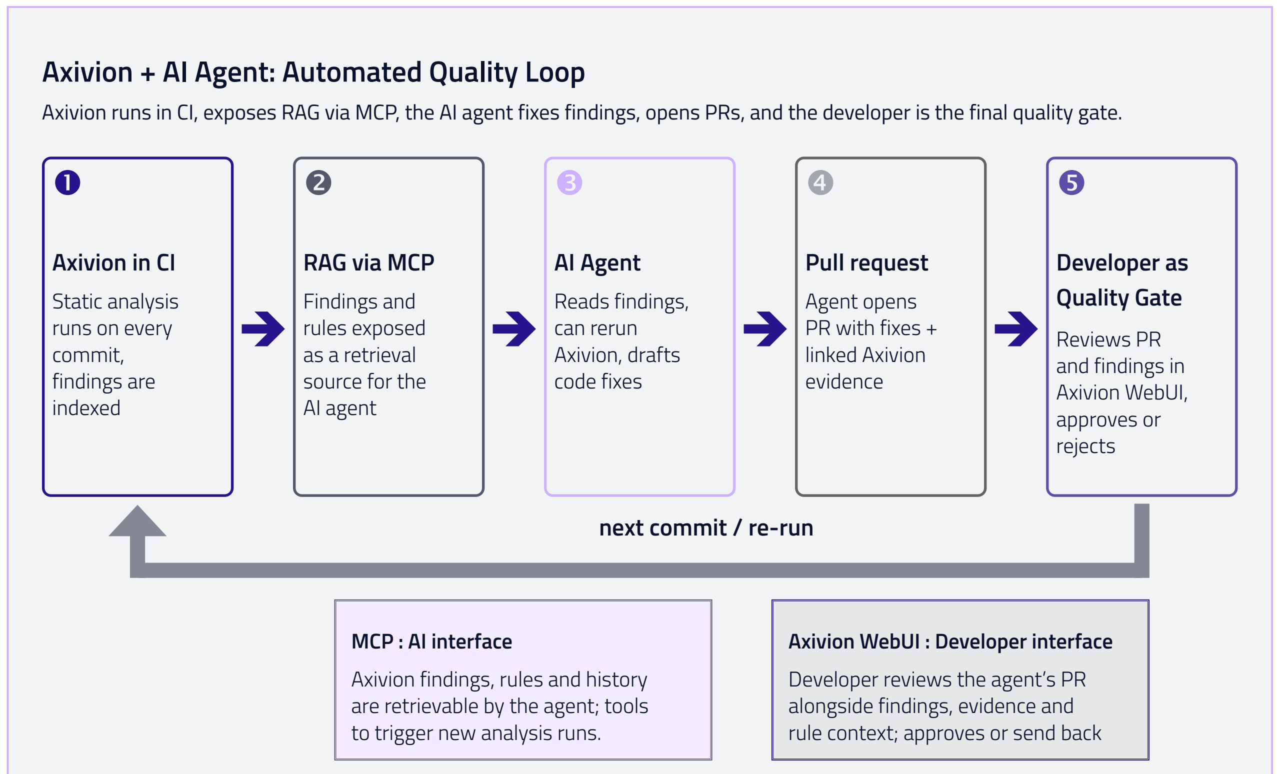
For an AI coding assistant, Axivion is a natural retrieval source: analysis results, rule rationales, architecture constraints, and historical findings are structured, current, and trustworthy. When the assistant is asked to fix a violation, refactor a module, or explain a finding, it can pull the relevant Axivion data and respond against the real state of the codebase — not a plausible approximation of it.



Two Levels of Integration: Interactive and Agentic Autonomous Use

Axivion connects to AI-assisted development at two levels of depth. The first is a protocol-level integration: an MCP server that exposes Axivion's analysis to any AI assistant or agent that knows how to speak MCP. The second is a deeper, IDE-native integration through the Axivion VS Code plugin. The two are complementary: the same analysis backs both, and most teams will use them together.

Level 1: MCP integration: Axivion as a tool for any AI assistant



The Model Context Protocol (MCP) is an open standard for letting AI assistants call external tools and pull external data at query time. An MCP server exposes a set of capabilities; any MCP-aware assistant, e.g. IDE-embedded copilots, chat assistants, autonomous agents, can connect, ask what tools and data are available, and use them. Axivion ships an MCP server that turns its analysis results and configuration into exactly this kind of consumable context.

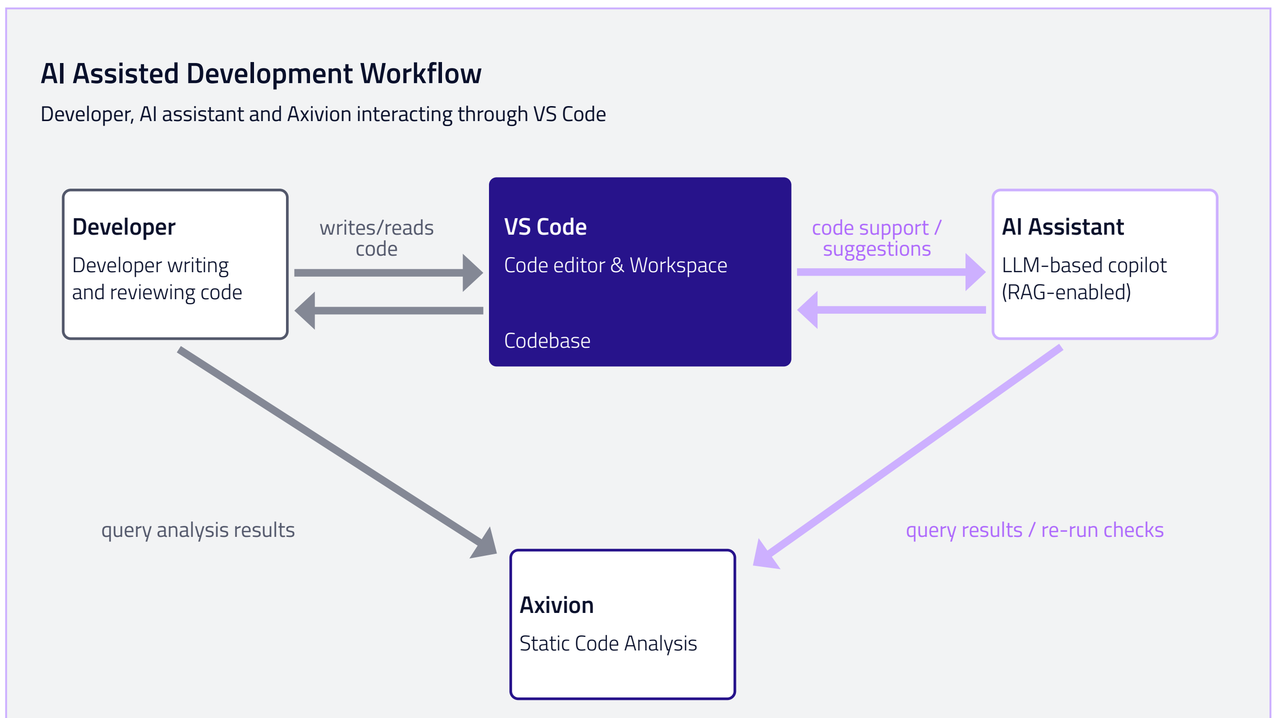
What the server exposes is the substance of the analysis, not a summary of it: the rules applied to the system, the issues reported for the code with their location and context, the rationale behind each rule and concrete hints for resolving the violation, and the project-specific configuration that shapes the analysis, such as third-party code, generated code, deviation workflows, and the project-specific conventions that an AI assistant could not otherwise infer. An assistant connected over MCP queries Axivion the way a developer would, but as structured tool calls rather than ad-hoc grep or file reads.

Because the protocol is generic, the same Axivion MCP server serves both interactive and autonomous use. A developer pairing with an AI assistant in their IDE gets the same grounded answers, e.g. “why is this flagged”, “what’s the fix”, “where is this called from”, that an agent running headless in CI gets when it triages findings, opens a pull request with proposed fixes, and routes the result back through Axivion’s quality gates for human approval. One server, one source of truth, two consumption patterns.

Level 2: The VS Code plugin: Axivion native in the IDE

The VS Code plugin embeds Axivion directly in the developer’s editor. It connects to the Axivion analysis results server, surfaces findings inline alongside the code they refer to, and lets the developer — and the AI assistant working in the same workspace — query analysis context and re-run analysis locally on a file, sub-project, or the whole project without leaving the IDE.

For an AI-assisted workflow, this closes the loop tightly. The assistant can read the current findings for the file in focus, see the rule rationale and suggested resolution, generate or rewrite code with that context in hand, and trigger a fresh local analysis to confirm the edit cleared the violation before the developer commits. Findings become a first-class input to AI-assisted coding rather than something checked after the fact, and the developer keeps the same review-and-approve role they have for any other change.



Best Practices:
Working With AI and Axivion

Run Axivion on every AI-generated change

Make analysis a non-negotiable step in the path from generation to commit. If an AI assistant writes or rewrites a file, Axivion runs against the result before the change is reviewable: locally through the IDE plugin, in the merge pipeline, or both. The point is not extra ceremony; it is that the verification regime does not depend on who or what produced the lines. Without this, throughput gains from AI translate directly into a larger surface of unverified code.

Feed Axivion findings into your prompts

An AI assistant prompted with the current Axivion findings for a file produces materially better edits than one prompted blind. Rule IDs, locations, rationales, and suggested resolutions are exactly the kind of structured, project-specific context an LLM cannot infer from surrounding tokens. The MCP integration makes this automatic. The assistant queries Axivion as a tool but the same principle holds when context is pasted manually: a prompt that names the violated rule and quotes the rationale beats one that does not.

Prompt with violation context, not blind

A bad prompt asks the assistant to "fix this function" with no further context. A good prompt names the specific Axivion finding (e.g. rule, location, rationale) and asks for a fix that resolves it without introducing new violations. The difference shows up in the result: the first produces a plausible rewrite that may or may not pass analysis; the second produces a targeted change against a concrete rule. Treat findings as the starting point of the conversation with the assistant, not an after-the-fact filter.

Constrain the assistant with explicit rules

AI assistants follow the rules they are given, and the absence of rules is itself a rule; one that lets the model fall back on whatever pattern looks most plausible. The fix is to make the constraints explicit, every time. Tell the assistant which coding standard applies (MISRA, AUTOSAR C++14, CERT, the project's own rule set etc.). Tell it not to introduce new violations while resolving an existing one, not to suppress findings, and not to invent APIs, types, or functions that are not in the codebase. Tell it that when Axivion and its own judgment disagree, Axivion is authoritative. A finding is not a suggestion to weigh against the model's confidence. The MCP integration enforces some of this by construction, because the assistant can query the rule set and current findings directly rather than guess at them, but the prompt still has to set expectations: ground the change in retrieved Axivion context, do not work around findings, and surface uncertainty to the developer rather than paper over it. A few sentences of this discipline at the start of a session changes the character of everything the assistant produces afterwards.

Do not relax quality rules for AI code

It is tempting, when an assistant generates code that trips a rule, to suppress the finding or carve out an exception. Resist this. The rule set encodes the project's safety, portability, and maintainability commitments; weakening it for AI output trades a short-term unblock for a long-term erosion of the evidence base that functional-safety standards require. If a rule is genuinely wrong for the project, change it deliberately and document the decision. Do not loosen it ad hoc because the assistant could not satisfy it.

Know what each side catches

AI assistants and Axivion are good at different things, and recognising the split is what makes the combination effective. AI assistants are strong at local pattern completion and at translating intent into syntax, drafting and refactoring across familiar idioms at a speed no

human matches. What they do not do is reason reliably about the whole project: they can miss cross-module side effects, invent APIs that do not exist, and reproduce patterns that violate coding standards or architecture rules they were never told about. Axivion is deterministic and global. It checks rule conformance, architecture conformance, data-flow correctness, and dead code reliably across the entire project, with the same result every run; the kind of repeatable, auditable evidence that functional-safety standards require. What static analysis does not do is verify intent: whether the behaviour the code implements is the behaviour the user actually wants. That judgment stays with the developer. Use each for what it is best at: AI to draft and refactor, Axivion to verify and constrain, the developer to decide.

The Developer Is Still in Charge

In safety-regulated development, a human in the loop is not a best practice — it is a requirement. Functional-safety standards assign responsibility and liability to people and organisations, not to tools. An AI assistant can draft, refactor, and accelerate, but it cannot sign off on a release, defend a design decision in an audit, or carry the legal weight of a field failure. Those obligations stay with the developer and the engineering organisation behind them.

The workflow described in this paper reinforces that position rather than diluting it. Axivion enforces the same rules against AI-generated code as against human-written code; the MCP integration grounds the assistant in the project's real findings and constraints; the IDE plugin keeps the developer in the review-and-approve seat for every change. AI handles more of the mechanical work, and the developer's attention shifts upward: to intent, architecture, and the judgment calls that determine whether the code does what the system actually needs to do.

The outcome is not a smaller role for the developer. It is a more leveraged one. With deterministic verification on one side and probabilistic generation on the other, the engineer remains the arbiter: deciding what to build, accepting what to ship, and owning the result.

Conclusion

AI-assisted development is not a future state; it is already the daily reality for most engineering teams. The question is no longer whether to use AI tools, but how to use them without eroding the quality and compliance standards that safety-critical development requires.

The answer this paper has set out is straightforward. Treat AI-generated code exactly as you would treat human-written code: run the same analysis, apply the same rules, require the same evidence. Use Axivion as the deterministic backbone that grounds AI suggestions in the real state of your codebase: through MCP for any AI assistant or agent, through the VS Code plugin for IDE-native workflows, and through CI gates that do not distinguish between human and machine authorship.

Where AI introduces velocity, Axivion provides the verification that makes that velocity safe to act on. Where AI introduces uncertainty (such as hallucinated APIs, standard violations, architectural drift) static analysis and architecture verification catch it before it reaches a human reviewer or a customer.

The developer remains the decision-maker throughout. Functional-safety standards require it, and good engineering judgment demands it. What changes is the leverage available: more code analysed, more findings surfaced earlier, more time freed for the architectural and intent-level decisions that tools cannot make.

Used together, AI generation and deterministic verification produce something neither achieves alone: development that is both faster and more defensible.

www.qt.io/axivion-ai

©Qt Group | 20260605